CECS 274: Data Structures Introduction

Oscar Morales Ponce

California State University Long Beach

Are data structures important to learn?

- Data structures improve our quality of life
- Many multi-million and several multi-billion dollar companies have been built around data structures
  - Open a file. File system data structures are used to locate the parts of that file on disk so they can be retrieved
  - Look up a contact on your phone
  - Log in to your favorite social network
  - Do a web search
  - Phone emergency services (9-1-1)

The Need for Efficiency

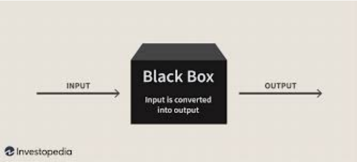
- Data can be stored in an array, iterate over all the elements to search an item and possibly adding or removing an element
- This implementation is straightforward, but not very efficient. Does this really matter?
- Computers are becoming faster and faster
  - Imagine an application with a moderately-sized data set, say of one million (10^6) items.
  - An application can look up each item at least one
  - Then at least $(10^6)(10^6) = 10^{12}$ inspections
  - Processor speeds. No more than 10^9 operations per second (1 gigahertz =billion of cycles per second)
  - It will take at least $10^{12}/10^9 = 1000$ seconds

Bigger data sets

- Google indexes over 8.5 billion web pages
- Any query over this data would take at least 8.5 seconds
- Google receives over 4,500 queries per second
- They would require $4,500(8.5) = 38,250$ very fast servers
- The solution is to carefully organize data within the data structure so that not every operation requires every data item to be inspected.

Interfaces

- Difference between an interface and its implementation
  - Interface: describes **WHAT** a data structure does
  - Implementation: describes **HOW** the data structure does it
- An interface, **ADT (Abstract Data Type)** defines the set of operations supported by a data structure and the semantics, or meaning, of those operations.
- An interface only provides a list of supported operations along with specifications about what types of arguments each operation accepts and the value returned by each operation.



Implementation

- A data structure implementation includes the internal representation of the data structure as well as the definitions of the algorithms for each operations supported.
- There can be many implementations for a single interface

The Queue Interfaces

- The **Queue** interface represents a collection of elements to which we can add elements and remove the next element.
- The operations supported by the Queue interface are
  - **add(x)**: add the value x to the Queue
  - **remove()**: remove the next (previously added) value, y, from the Queue and return y
  - **size()**: the number of items in the list Queue
- The Queue's queueing discipline decides which element should be removed.

FIFO (first-in-first-out) Queue

- Removes items in the same order they were added
- This is the most common kind of Queue so the qualifier FIFO is often omitted.
- $add(x)$ and $remove()$ operations on a FIFO Queue are often called $enqueue(x)$ and $dequeue()$, respectively.

Priority Queue

- Always removes the smallest element from the Queue, breaking ties arbitrarily
- The $remove()$ operation on a priority Queue is usually called $delete\_min()$ in other texts.

LIFO (last-in-first-out) Queue or Stack

- The most recently added element is the next one removed
- This structure is so common that it gets its own name: Stack
- Often $add(x)$ and $remove()$ are changed to $push(x)$ and $pop()$

Dequeu (Generalization of both the FIFO Queue and LIFO Queue)

- Elements can be added at the front of the sequence or the back of the sequence.
- The names of the Deque operations are self-explanatory: $add\_first(x), remove\_first(), add\_last(x)$, and $remove\_last()$
- A Stack can be implemented using only $add\_first(x)$ and $remove\_first()$
- A FIFOQueue can be implemented using $add\_last(x)$ and $remove\_first()$


+


The List Interface

- The List interface includes the following operations:
  - **size()**: return n, the length of the list
  - **get(i)**: return the value x_i
  - **set(i, x)**: set the value of x_i equal to x
  - **add(i, x)**: add x at position i, displacing $x_i, ..., x_{n-1}$; Set $x_{j+1} = x_j$, for all $j \in n-1, ..., i$, increment n, and set $x_i = x$
  - **remove(i)** remove the value x_i, displacing $x_{i+1}, ..., x_{n-1}$; Set $x_j = x_{j+1}$, for all $j \in n-1, ..., i$, decrement n
  - **append(x)**: add x at position n; Increment n, and set $x_{n-1} = x$

Implementing the Deque interface using a list interface

- $add\_first(x) \rightarrow add(0, x)$
- $remove\_first() \rightarrow remove(0)$
- $add\_last(x) \rightarrow add(size(), x)$
- $remove\_last() \rightarrow remove(size() - 1)$

The $USet$ Interface: Unordered Sets

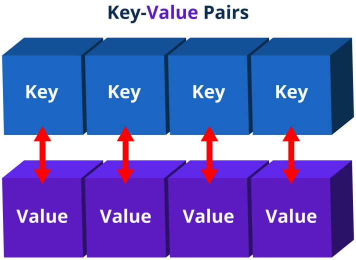
- The $USet$ interface represents an unordered set of unique elements, which mimics a mathematical set
- A $USet$ contains n distinct elements
- No element appears more than once
- The elements are in no specific order.

The $USet$ opperations: Unordered Sets

- A $USet$ supports the following operations
  - **size()**: return the number, n, of elements in the set
  - **add(x)**: add the element x to the set if not already present;
    - Add x to the set provided that there is no element y in the set such that x equals y.
    - Return true if x was added to the set and false otherwise.
  - **remove(x)**: remove x from the set;
    - Find an element y in the set such that x equals y and remove y.
    - Return y, or nil if no such element exists.
  - **find(x)**: find x in the set if it exists;
    - Find an element y in the set such that y equals x.
    - Return y, or nil if no such element exists.

Dictionary/Map

- To create a dictionary/map, one forms compound objects called Pairs, each of which contains a key and a value
- Two Pairs are treated as equal if their keys are equal
- If we store some pair (k, v) in a $USet$ and then later call the $find(x)$ method using the pair $x = (k, nil)$ the result will be $y = (k, v)$

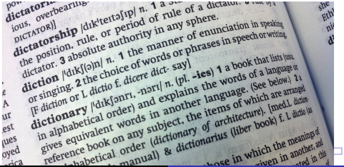


The SSet Interface: Sorted Sets

- An SSet stores elements from some total order, so that any two elements x and y can be compared with the method

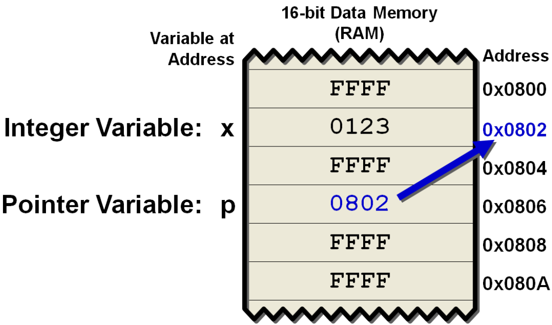
$$compare(x,y) \begin{cases} < 0 & \text{if } x < y \\ > 0 & \text{if } x > y \\ = 0 & \text{if } x = y \end{cases}$$

- An SSet supports the $size(), add(x)$, and $remove(x)$ methods with exactly the same semantics as in the $USet$ interface.
  4. **find(x)**: locate x in the sorted set;
     Find the smallest element y in the set such that $y \geq x$.
     Return y or nil if no such element exists.



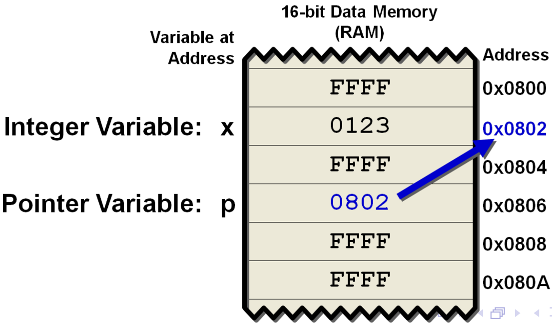
Mode of computation

- We use the ω-bit word-RAM model (Random Access Machine)
- In this model, we have access to a random access memory consisting of cells, each of which stores a ω-bit word.
- This implies that a memory cell can represent, for example, any integer in the set $\{0, ..., 2^{\omega-1}\}$



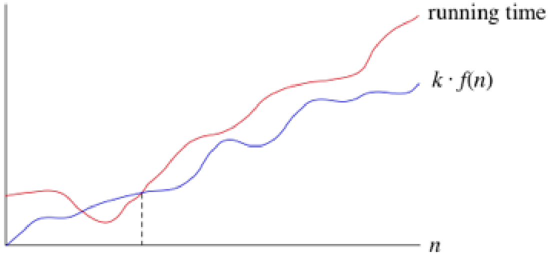
Mode of computation 2

- In the word-RAM model, basic operations on words take constant time (arithmetic, comparison, bitwise operations.
- Any cell can be read or written in constant time
- Allocating a block of memory of size k takes $O(k)$ time and returns a reference (a pointer) to the newly-allocated memory block.
- Space is measured in words



Correctness, Time Complexity, and Space Complexity

- **Correctness**: The data structure should correctly implement its interface.
- **Time complexity**: The running times of operations on the data structure should be as small as possible.
- **Space complexity**: The data structure should use as little memory as possible.



Three different kinds of running time guarantees

- **Worst-case running times**: The operation takes at most $O(f(n))$ for any input of size n
- **Amortized running times**: After a sequence of m operations the time it takes is at most $O(mf(n))$ for any input of size n
- **Expected running times**: The operation is **expected** to take at most $O(f(n))$ for any input of size n