Outline

- Dictionaries
- Binary Search
- Recurrence equations
Dictionary ADT

- Dictionary ADT models a searchable collection of **key-value** pairs \((k,v)\), called **entries**
  - dictionary stores entries, entries are located using keys
  - keys and values can be of any type
  - keys may be ordered or unordered

- Applications:
  - word-definition pairs
  - bank account number-customer information
  - Internet IP address-host names
    - 128.148.34.101-www.datastructures.net

- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key can be allowed/not allowed
Our dictionary will make an explicit use of entries (the key-value pairs stored in a dictionary)

Assume each entry comes equipped with `key()` and `value()` methods to respectively access its key and value components
Dictionary ADT

Dictionary ADT methods:

- **find**(k): if the dictionary has an entry with key k, return the entry (k,v), else, returns null
- **findAll**(k): returns an iterator of all entries with key k
- **insert**(k,v): inserts key-value entry (k,v) and returns the entry created
- **remove**(e): remove the entry e, returning the removed entry e or null if e is not in the dictionary
- **entries()**: returns an iterator of the entries in the dictionary
- **size()**: return the number of items
- **isEmpty()**: test whether dictionary is empty
### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5,A)</td>
<td>(5,A)</td>
<td>(5,A)</td>
</tr>
<tr>
<td>insert(7,B)</td>
<td>(7,B)</td>
<td>(5,A),(7,B)</td>
</tr>
<tr>
<td>insert(2,C)</td>
<td>(2,C)</td>
<td>(5,A),(7,B),(2,C)</td>
</tr>
<tr>
<td>insert(8,D)</td>
<td>(8,D)</td>
<td>(5,A),(7,B),(2,C),(8,D)</td>
</tr>
<tr>
<td>insert(2,E)</td>
<td>(2,E)</td>
<td>(5,A),(7,B),(2,C),(8,D),(2,E)</td>
</tr>
<tr>
<td>find(7)</td>
<td>(7,B)</td>
<td>(5,A),(7,B),(2,C),(8,D),(2,E)</td>
</tr>
<tr>
<td>find(4)</td>
<td>null</td>
<td>(5,A),(7,B),(2,C),(8,D),(2,E)</td>
</tr>
<tr>
<td>find(2)</td>
<td>(2,C)</td>
<td>(5,A),(7,B),(2,C),(8,D),(2,E)</td>
</tr>
<tr>
<td>findAll(2)</td>
<td>(2,C),(2,E)</td>
<td>(5,A),(7,B),(2,C),(8,D),(2,E)</td>
</tr>
<tr>
<td>size()</td>
<td>5</td>
<td>(5,A),(7,B),(2,C),(8,D),(2,E)</td>
</tr>
<tr>
<td>remove(find(5))</td>
<td>(5,A)</td>
<td>(7,B),(2,C),(8,D),(2,E)</td>
</tr>
<tr>
<td>find(5)</td>
<td>null</td>
<td>(7,B),(2,C),(8,D),(2,E)</td>
</tr>
</tbody>
</table>
Two Types of Dictionaries

1. Unordered dictionary
   - keys do not have to be ordered
   - can test keys for equality
   - can be based on linked list (called log-file or audit trail)

2. Ordered dictionary
   - keys do have to be ordered
   - can compare keys
   - can be based on array (called search table)
A List-Based Dictionary

- A log file or audit trail is a dictionary implemented by means of unordered or unsorted sequence
  - store items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
Linked List Methods

- **first()**: returns position of the first element in list $S$
- **next($p$)**: return the position of the element in $S$ following the element at position $p$
- **insertLast($e$)**: Insert a new element $e$ into list $S$ as the last element in $S$
- **remove($p$)**: remove from $S$ the element at position $p$
Iterators

- **Iterator** is an ADT for scanning through a collection of elements, one element at a time
- Has notion of *current* element
- Supports two methods
  - `hasNext()`
    - test whether there are elements left in the iterator
  - `next()`
    - return the next element in the iterator
    - *current* element is advanced
    - first call to `next()` returns the first element in the iterator
Linked List Iterators

- We assume our linked list supports methods
  - `positions()`: returns an iterator of positions in the list
  - `elements()`: returns an iterator of elements stored in the list
Algorithm entries()

Input: None
Output: Iterator of entries in $D$

return $D$.elements()  // elements of linked list $D$ are the entries in our dictionary
Algorithm find(k)

**Input:** A key k  
**Output:** return any entry with key equal to k

B = D.elements()  
while B.hasNext() do  
  e = B.next()  // next entry in the iterator  
  if e.key() = k then  
    return e   
return null
The findAll(k) Algorithm

Algorithm findAll(k)

Input: A key k

Output: Iterator of all entries with key k

Create an initially-empty list L
B = D.elements()

while B.hasNext() do
    e = B.next()
    if e.key() = k then
        L.insertLast(e)

return L.elements()
Algorithm insert(k,v)

**Input:** A key k and value v

**Output:** entry (k,v) is added into D

Create a new entry e = (k,v)

//D is linked-list storing dictionary entries

D.insertLast(e)

return e
Algorithm remove(e)

**Input:** entry e

**Output:** The removed e, or null if e was not in D

B = D.positions()

while B.hasNext() do
  p = B.next()
  if p.element() = e then
    D.remove(p)
    return e
  return null  //there is no entry with key equal to k
Performance of List-Based Dictionary

- **insert** takes $O(1)$ time since we insert the new item at the end of the sequence.

- **find** and **remove** take $O(n)$ time since in the worst case traverse entire sequence to look for an item with the given key.

- Space requirement: $O(n)$

- Unsorted list implementation is effective only for
  - dictionaries of small size
  - for dictionaries on which insertions are the most common operations, while searches and removals are rare
    - ex: historical record of logins to a workstation
Ordered Dictionary (Array-based)

- Also called an **ordered search table**
- Entries are stored in non-increasing order

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>13</td>
<td>15</td>
<td>25</td>
<td>105</td>
</tr>
</tbody>
</table>
Ordered Dictionary (Array-based)

- Perform usual dictionary operations, but also maintain an order relation for the keys.
- In addition, can also support methods:
  - `first()`: return the entry with the smallest key.
  - `last()`: return the entry with the largest key.
  - `successors(k)`: return iterator of entries with keys greater than or equal to k, in nondecreasing order.
  - `predecessors(k)`: return iterator of entries with keys less than or equal to k, in nondecreasing order.
- For general implementation, use a *comparator* to provide the order relation among the keys.
Comparator for Key Comparison

- Can require keys know how to compare themselves
  - public class Key implements Comparable Interface;
- Sometimes keys do not know how to they ought to be compared
  - Suppose key is an (x, y) coordinate
  - There is no inherent way to compare coordinates
    - (3,5) < (5,4) or (3,5) > (5,4) ?
  - Sometimes need comparisons based only on the x coordinate, sometimes on the y coordinate, and other ways to compare might be needed
Comparator

- For most general approach, should not rely on keys to provide their comparison rules
- Define special **comparator** objects
  - object that supplies rules to compares two keys
  - external to the keys
- Give dictionary $D$ the comparator when $D$ is constructed
Comparator ADT

- Has a single method that takes two keys and compares them
  - `compare(a, b)`

  input: a pair of objects, output: integer \( i \) s.t.

  \[
  \begin{align*}
  i < 0 & \quad \text{if } a < b \\
  i > 0 & \quad \text{if } a > b \\
  i = 0 & \quad \text{if } a = b
  \end{align*}
  \]

```java
public interface Comparator<T>{
    public int compare(T a, T b);
}
```
public class Xbased implements Comparator<Point2D> {
    int xa, ya, xb, yb;

    public int compare(Point2D a, Point2D b) {
        xa = a.getX();
        xb = b.getX();
        return (xa - xb);
    }
}
public class OrderedDictionary<K,V> implements DictionaryInterface<K,V> {
    ArrayList<Entry<K,V>> S;
    protected Comparator<K> c;

    public OrderedDictionary(Comparator<K> comp) {
        c = comp;
        S = new ArrayList<Entry<K,V>>(());
    }

    public Entry findKey(K k) {
        KeyIterator B = S.entries();
        Entry<K,V> e;

        while (B.hasNext()) {
            e = B.next();
            if (c.compare(e.getKey(), k) == 0)
                return e;
        }
        return null;
    }
}
Array-Based Dictionary Performance

- **Advantage**
  - Searching is $O(\log n)$ using **binary search** algorithm

- **Disadvantage**
  - Inserting and removing are $O(n)$, need to shift elements to maintain the ordering of keys

- A search table is effective only for
  - dictionaries of small size
  - for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations, language dictionary)
Binary Search

- Performs `find(k)` on an ordered dictionary, implemented as array-based sequence, sorted by key
  - at each step, the number of candidate items is halved
- Example: `find(8)`

```
| 0 | 3 | 4 | 7 | 8 | 9 | 12 | 14 | 17 | 21 | 22 | 26 | 29 |

8 < m
```

```
| 0 | 3 | 4 | 7 | 8 | 9 | 12 | 14 | 17 | 21 | 22 | 26 | 29 |

8 > m
```

```
| 0 | 3 | 4 | 7 | 8 | 9 | 12 | 14 | 17 | 21 | 22 | 26 | 29 |

8 = m
```
Pseudo-Code for Binary Search

Algorithm BinarySearch(S,k,low,high)

if low > high then
    return NO_SUCH_KEY
else
    mid = (low+high)/2
    if k == key(mid) then
        return key(mid)
    else if k < key(mid) then
        return BinarySearch(S,k,low,mid-1)
    else
        return BinarySearch(S,k,mid+1,high)
Worst Case Analysis for Binary Search

- Let \( T(n) \) be time complexity when array size is \( n \)
- Base case, \( n = 0 \): constant number \( c \) of operations performed
  \[ T(0) = c \]
- Recursive case, \( n > 0 \): constant number \( k \) of operations plus operations performed in recursive call on array of size \( (n - 1)/2 \)
  \[ T(n) = k + T((n - 1)/2) \]
- System of recurrence equations that describe complexity:
  \[
  \begin{align*}
  T(0) &= c \quad \text{// base case} \\
  T(n) &= k + T((n - 1)/2) \quad \text{// recursive case}
  \end{align*}
  \]
Solving Recurrence Equations

- *Unwrap* recursion until pattern emerges
- Unwrap the first time

\[
T\left(\frac{n-1}{2}\right) = k + T\left(\frac{n-1}{2} - 1\right) = k + T\left(\frac{n-1-2}{2}\right) = k + T\left(\frac{n-2^0 - 2^1}{2^2}\right)
\]

- Result after unwrapping once

\[
T(n) = 2k + T\left(\frac{n-2^0 - 2^1}{2^2}\right)
\]

\[
T(0) = c
T(n) = k + T\left(\frac{n-1}{2}\right)
\]
Solving Recurrence Equations

- Result after unwrapping once
  \[ T(n) = 2k + T\left(\frac{n - 2^0 - 2^1}{2^2}\right) \]

- Unwrap the second time
  \[ T\left(\frac{n - 2^0 - 2^1}{2^2}\right) = k + T\left(\frac{n - 2^0 - 2^1}{2^2} - 1\right) = k + T\left(\frac{n - 2^0 - 2^1 - 2^2}{2^3}\right) \]

- Result after unwrapping twice
  \[ T(n) = 3k + T\left(\frac{n - 2^0 - 2^1 - 2^2}{2^3}\right) \]
Solving Recurrence Equations

- Result after unwrapping twice

\[ T(n) = 3k + T \left( \frac{n - 2^0 - 2^1 - 2^2}{2^3} \right) \]

- Could keep unwrapping, but the pattern is clear now

\[ T(n) = ik + T \left( \frac{n - 2^0 - 2^1 - \ldots - 2^{i-1}}{2^i} \right) = ik + T \left( \frac{n - \left(2^0 + 2^1 + \ldots + 2^{i-1}\right)}{2^i} \right) \]

- Compute:

\[ S = 2^0 + 2^1 + 2^2 + \ldots + 2^{i-1} \]

\[ 2S = 2^1 + 2^2 + 2^3 + \ldots + 2^{i-1} + 2^i \]

\[ S = 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{i-1} \]

\[ S = -2^0 + 2^i \]
Solving Recurrence Equations

\[ S = 2^0 + 2^1 + 2^2 \ldots + 2^{i-1} = 2^i - 1 \]

- The pattern becomes

\[ T(n) = ik + T \left( \frac{n - \left(2^0 + 2^1 + \ldots + 2^{i-1}\right)}{2^i} \right) = ik + T \left( \frac{n - 2^i + 1}{2^i} \right) \]

- Stop expanding recursion when reach the base case \( T(0) \)

- This happens when \( \frac{n - 2^i + 1}{2^i} = 0 \)

- Solving for \( i \)

\[ n - 2^i + 1 = 0 \Rightarrow 2^i = n + 1 \Rightarrow i = \log(n + 1) \]

- Plugging in \( i \)

\[ T(n) = ik + T \left( \frac{n - 2^i + 1}{2^i} \right) = k \log(n+1) + T(0) = k \log(n+1) + c \]

- Conclusion: \( T(n) \) is \( O(\log n) \)
## Log File vs Search Table

<table>
<thead>
<tr>
<th>Method</th>
<th>Log File</th>
<th>Search Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>keys, elements</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>findElement</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>findAllElements</td>
<td>O(n)</td>
<td>O(log n+s)</td>
</tr>
<tr>
<td>insertItem</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>removeElement</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>removeAllElements</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Dictionaries

- What if we need frequent insert, find, remove?
- Would like find, insert, remove to be $O(1)$
  - with hash tables (study next) average time is $O(1)$
  - worst case time is still linear
- Later will study balance search trees data structure
  - find, insert, remove is $O(\log n)$ in the worst case