Dictionaries

The Associative Array

CSCI 3700 — Data Structures and Objects

Department of Computer Science and Information Systems
Youngstown State University

Robert W. Kramer

Outline

Motivation

"Uncle Owen, this R2 unit has a bad motivator, look!" — Luke Skywalker

Consider an array. . .

- Collection of values
- Each value assigned an numeric index

What if a number isn't an appropriate index?

- Players identified by position
- Song music / lyrics identified by title

Dictionaries enable other types of indexing

Key-Value Pairs

Dictionary data is stored using *key-value* pairs

- Key
  - The identifier used to access values
  - The "index"
- Value
  - The datum that is stored / retrieved
  - The "content"

Parallel Arrays

An alternative to an array of structures (1/3)

Suppose we have a structure with two fields, *key* and *val*

Now, consider an array of such structures

dictionary	key: k_0 val: v_0	key: k_1 val: v_1	key: k_2 val: v_2	$\cdots$	key: k_{n-2} val: v_{n-2}	key: k_{n-1} val: v_{n-1}

Can access pair i with **dictionary[i].key** and **dictionary[i].val**

Parallel Arrays

An alternative to an array of structures (2/3)

Can also use two *parallel arrays* to store keys and values:

keys	k_0	k_1	k_2	$\cdots$	k_{n-2}	k_{n-1}
values	v_0	v_1	v_2	$\cdots$	v_{n-2}	v_{n-1}

keys[i] and **values[i]** are key-value pair *i*

Parallel Arrays

An alternative to an array of structures (3/3)

Why use parallel arrays?

- More efficient memory allocation(?)
- May be easier to implement than structures
- Syntactically simpler

The Dictionary ADT

Like a conventional dictionary, only more so

A *Dictionary* is a container that supports the following operations:

- insert(k, v)
  - Insert the key-value pair $k - v$ into the dictionary
  - Key k must be unique
- remove(k)
  - Remove key-value pair with key k from dictionary
- search(k)
  - Search for key k, return k's value
- update(k, v)
  - Update existing key k's value to v

Key and Value Limitations

Keys have minor limitations

- Must be comparable with ==
- In sorted implementation, must also be comparable with <

Values have no limitations

Dictionary Implementations

Choosing a backing store (1/2)

Three methods for storing container data:

- An array
- A linked structure
- A simulated linked structure

For this discussion, we will use parallel arrays

Dictionary Implementation

Data arrangement options (2/2)

Three methods for arranging dictionary data within the backing store:

- Unsorted array
- Sorted array
- Hash table

We will examine each of these options

Unsorted Insertion

General approach (1/2)

The general approach:

1. Perform SequentialSearch to find the key
2. If the search is successful, throw an exception
   - Keys must be unique!
3. If the the search fails, add key and value to end of list

Unsorted Insertion

The algorithm (2/2)

```
procedure UnsortedInsert(k, v)
    i ← 0
    while i < n and keys[i] ≠ k do
        i ← i + 1
    end while
    if i < n then
        throw DuplicateKeyException(k)
    else
        keys[i] ← k
        values[i] ← v
        n ← n + 1
    end if
end procedure
```

Unsorted Removal

The approach, in pictures (1/2)

First, search for the key:

					key					
keys	k_0	k_1	$\cdots$	k_{i-1}	k_i	k_{i+1}	$\cdots$	k_{n-2}	k_{n-1}

Next, subtract 1 from n:

keys	k_0	k_1	$\cdots$	k_{i-1}	k_i	k_{i+1}	$\cdots$	k_{n-1}	k_n

Finally, copy the key and value from position n into position i:

keys	k_0	k_1	$\cdots$	k_{i-1}	k_n	k_{i+1}	$\cdots$	k_{n-1}	k_n

Unsorted Removal

The algorithm (2/2)

procedure $\textsc{UnsortedRemove}(k)$
    $i \leftarrow 0$
    while $i < n$ **and** $keys[i] \neq k$ **do**
        $i \leftarrow i + 1$
    end while
    if $i < n$ **then**
        $n \leftarrow n - 1$
        $keys[i] \leftarrow keys[n]$
        $values[i] \leftarrow values[n]$
    else
        throw KeyNotFoundException(k)
    end if
end procedure

Unsorted Search and Update

Even closer to SequentialSearch (1/3)

Search is exactly SequentialSearch

Update is very similar

- On successful search, store new value in position *i*
- No value returned

Both throw an exception if key is not found

Unsorted Search

The search algorithm (2/3)

procedure UnsortedSearch(k)
    $i \leftarrow 0$
    while $i < n$ **and** $keys[i] \neq k$ **do**
        $i \leftarrow i + 1$
    end while

    if $i < n$ **then**
        return $values[i]$
    else
        throw KeyNotFoundException(k)
    end if
end procedure

Unsorted Update

The update algorithm (3/3)

procedure $\textsc{UnsortedUpdate}(k, v)$
    $i \leftarrow 0$
    while $i < n$ **and** $keys[i] \neq k$ **do**
        $i \leftarrow i + 1$
    end while

    if $i < n$ **then**
        $values[i] \leftarrow v$
    else
        throw KeyNotFoundException(k)
    end if
end procedure

Analysis of Operations

How much time do the operations need?

All operations utilize a sequential search

- Keys are unordered, so no better search can be used

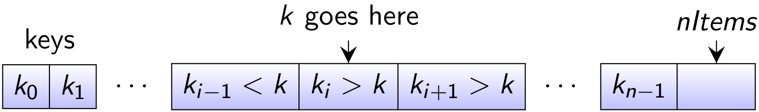
The four Dictionary operations are in $O(n)$

The common operations are all in $O(1)$

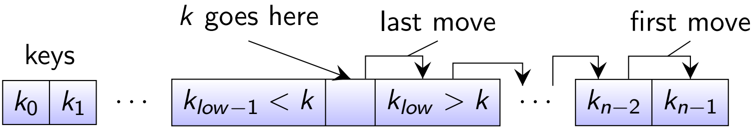
Sorted Insert
The process (1/2)

List must always be sorted!

Given a key-value pair k, v:



Move elements $> k$ over to make room



Sorted Insert

The algorithm (2/2)

procedure SortedInsert(v, k)

⋮ ▷ Same as ForgetfulBinarySearch through end of loop

  if $keys[low] = k$ **then**

    throw DuplicateKeyException(k)

  else

    $j \leftarrow n - 1$ ▷ Move larger keys over

    while $j >= 0$ **and** $keys[j] > k$ **do**

      $keys[j+1] \leftarrow keys[j]$

      $values[j+1] \leftarrow values[j]$

    end while

    $keys[j+1] \leftarrow k$ ▷ Key and value go here

    $values[j+1] \leftarrow v$

    $n \leftarrow n + 1$

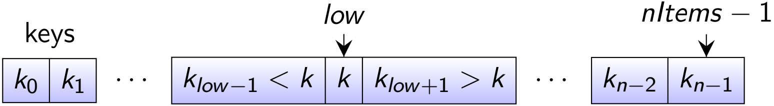
  end if

end procedure

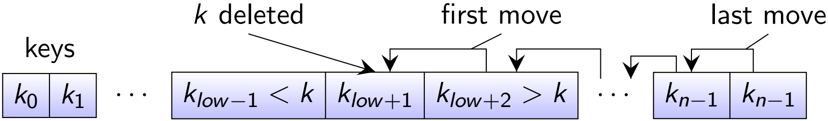
Sorted Removal

The process (1/2)

Use binary search to find k:



Move elements $> k$ over to cover k and its value:



Sorted Removal

The algorithm (2/2)

```
procedure SortedInsert(v, k)
    ⋮                          ▷ Same as ForgetfulBinarySearch through end of loop
    if keys[low] = k then
        n ← n − 1
        for j ← low to n − 1 do
            keys[j] ← keys[j + 1]
            values[j] ← values[j + 1]
        end for
    else
        throw KeyNotFoundException(k)
    end if
end procedure
```

Sorted Search and Update

The process (1/3)

Similar to process for unsorted dictionaries

- Search is just FORGETFULBINARYSEARCH
- Update is similar
  - Store new value instead of returning it
- Both throw exception if key not found

Sorted Search

The algorithm (2/3)

```
procedure SortedSearch(k)
    low ← 0
    high ← n − 1
    while low < high do
        mid ← (low+high)/2
        if keys[mid] < k then
            low ← mid + 1
        else
            high ← mid
        end if
    end while
    if keys[low] = k then
        return values[low]
    else
        throw KeyNotFoundException(k)
    end if
end procedure
```

Sorted Update

The algorithm (3/3)

```
procedure SortedUpdate(k, v)
    low ← 0
    high ← n − 1
    while low < high do
        mid ← (low+high)/2
        if keys[mid] < k then
            low ← mid + 1
        else
            high ← mid
        end if
    end while
    if keys[low] = k then
        values[low] ← v
    else
        throw KeyNotFoundException(k)
    end if
end procedure
```

Sorted Dictionary Analysis

- Insert: $O(n)$
    - $O(n)$ to move elements
- Remove: $O(n)$
    - $O(\lg n)$ to search
    - $O(n)$ to move elements
- Search and update: $O(\lg n)$
- Common operations are still all in $O(1)$

A Problem

An issue with deletion from a hash table (1/2)

Start with a set of inserted keys that all collide:

		h	$h+1$	$h+2$	$h+3$	$h+4$	
keys	$\cdots$	k_0	k_1	k_2	k_3	k_4	$\cdots$

Now, delete k_1:

		h	$h+1$	$h+2$	$h+3$	$h+4$	
keys	$\cdots$	k_0	$\cancel{k_1}$	k_2	k_3	k_4	$\cdots$

Searching for k_4 fails if we stop at any gap:

		h	$h+1$	$h+2$	$h+3$	$h+4$	
keys	$\cdots$	k_0	$\cancel{k_1}$	k_2	k_3	k_4	$\cdots$
probes:		$\uparrow$ 1^{st}	$\uparrow$ 2^{nd}				

A Problem

An issue with deletion from a hash table (2/2)

Searching for k_4 succeeds if we continue past deletions:

		h	$h+1$	$h+2$	$h+3$	$h+4$	
keys	$\cdots$	k_0	$\cancel{k_1}$	k_2	k_3	k_4	$\cdots$
probes:		1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	

Use a third array **status** to indicate whether a location is:

- unused
- in use
- previously used but deleted

Hash Dictionary Insert
The process (1/2)

All hash operations use the same basic process:

1. Hash k to get starting position
2. Probe until k found or proper gap found
3. Perform appropriate action

Note

This presentation uses linear probe

Hash Dictionary Insert

The algorithm (2/2)

```
procedure HashInsert(k, v)
    i ← Hash(k)

    while status[i] = IN_USE do
        if keys[i] = k then
            throw DuplicateKeyException(k)
        end if
        i ← (i + 1) mod TABLE_SIZE
    end while

    keys[i] ← k
    values[i] ← v
    status[i] ← IN_USE
end procedure
```

Hash Table Remove, Search and Update

All use almost identical process

Differ in how to proceed when key is found:

- Remove sets $status[i]$ to $DELETED$
- Search returns $values[i]$
- Update sets $values[i]$ to v

Hash Table Removal

```
procedure HashRemove(k)
    i ← Hash(k)

    while status[i] ≠ UNUSED do
        if status[i] = IN_USE and keys[i] = k then
            status[i] ← DELETED
            return
        end if
        i ← (i + 1) mod TABLE_SIZE
    end while

    throw KeyNotFoundException(k)
end procedure
```

Hash Table Search

```
procedure HashSearch(k)
    i ← Hash(k)

    while status[i] ≠ UNUSED do
        if status[i] = IN_USE and keys[i] = k then
            return values[i]
        end if
        i ← (i + 1) mod TABLE_SIZE
    end while

    throw KeyNotFoundException(k)
end procedure
```

Hash Table Update

procedure HashUpdate(k, v)
    $i \leftarrow Hash(k)$

    while $status[i] \neq UNUSED$ **do**
        if $status[i] = IN\_USE$ **and** $keys[i] = k$ **then**
            $values[i] \leftarrow v$
            return
        end if
        $i \leftarrow (i + 1) \bmod TABLE\_SIZE$
    end while

    throw KeyNotFoundException(k)
end procedure

Hash Dictionary Analysis

Remember the two assumptions about hashing:

- Keys are spread evenly by hash function
- Table doesn't get too full

If these assumptions are valid, then all hash dictionary operations are in $O(1)$

Summary

- Dictionaries store key-value pairs
- Three common methods for implementing backing store
- Three methods for implementing a dictionary
    - Unsorted list
    - Sorted list
    - Hash table