Ordered Array
Dynamic Array Implementation
Recall Arrays

• Operations
  – Add        $O(1+)$ for dynamic array
  – Contains   $O(n)$
  – Remove     $O(n)$

• What if our application needs to repeatedly call “Contains” to search for items (e.g. a phone book application)?
  – $O(n)$ would be too expensive.
Ordered Array Abstraction

- Same operations as Array
  - Add
  - Contains
  - Remove

- Property: elements in *sorted order*

- This gives us a chance to search more efficiently
Applications of Ordered Collections

- So search can be sped up with ordering
  - We’ll go into detail later

- What about other set operations?
  - Merging ordered collections?
    - Maintain order!
  - Set operations such as `union`, `intersection`, etc.
Fast Merge

5 9 10 12 17

1 8 11 20 32

5 9 10 12 17

× 8 11 20 32

5 9 10 12 17

× 8 11 20 32

5 9 10 12 17

× 8 11 20 32

5 9 10 12 17

× 8 11 20 32

1 5 8 9 10
Set Operations: Similar to Merge

• You can quickly merge two ordered arrays into a new ordered array
  – What is its complexity? \( \Rightarrow O(n) \)

• Set operations (intersection, union, difference, subset) are similar to merge
  – Try these on your own... (See Chapter 9)
**Fast Set Intersection**

<table>
<thead>
<tr>
<th>5 9 10 12 20</th>
<th>1 9 12 13 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 9 10 12 20</td>
<td>x 9 12 13 32</td>
</tr>
<tr>
<td>5 9 10 12 20</td>
<td>x 9 12 13 32</td>
</tr>
<tr>
<td>5 9 10 12 20</td>
<td>x 9 12 13 32</td>
</tr>
<tr>
<td>5 9 10 12 20</td>
<td>x 9 12 13 32</td>
</tr>
<tr>
<td>5 9 10 12 20</td>
<td>x 9 12 13 32</td>
</tr>
</tbody>
</table>

**Intersection:** 9 12
• We’ve already encountered the idea of searching an ordered list by repeatedly cutting it in half.

• The formal name for this process is **binary search**

• Each step cuts region containing the value in half

• Starting with *n* items, how many times can I cut in half before reaching a set of size one?
Binary Search: $O(\log n)$

- A $O(\log n)$ search is much much faster than an $O(n)$ search (for large $n$)
- $\log_2 1,000,000 \sim 20$
- Log of largest unsigned integer value in 32-bit machines: $\log_2(4294967295)$ is 32
  - Instead of 4 billion comparisons, you only need 32
What are the requirements for performing a binary search?

- Random access to the elements
- Elements are already in sorted order
Binary Search Ordered Array: Intuition

- Compute the middle index
- Check for the value at that index
- If found the value, done, return the index
- If not found
  - If value is less than value at the index, repeat with left half of array
  - Else repeat with right half of array

If value is not in array we would like to return index where it should be inserted to maintain order
/* this is an initial call into the recursive procedure */

int _binarySearch(TYPE * data, int size,
                   TYPE val)
{
    return recBinarySearch(data,0,size-1, val);
}
int _recBinarySearch(TYPE * data, int low, int high, TYPE val) {
    int mid;
    if (low <= high) {
        mid = (low + high) / 2;
        if (EQ(data[mid], val)) return mid;
        if (LT(data[mid], val))
            return _recBinarySearch(data, mid + 1, high, val);
        else
            return _recBinarySearch(data, low, mid - 1, val);
    }
    return low;
}
int _binarySearch(TYPE * data, int size, TYPE val) {
    int low = 0;
    int high = size;
    int mid;
    while (low < high) {
        mid = (low + high) / 2;
        if (EQ(data[mid], val))
            return mid;
        if (LT(data[mid], val))
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}
Binary Search Ordered Array: Return Value

• If value is found, returns index of that value

• If value is not found, returns position where it can be inserted without violating ordering

• NOTE: returned index can be larger than a legal index
Summary

• Searching DynArr and LinkedLists are $O(N)$ on average

• Binary Search provides $O(\log N)$ search but requires that
  – We have random access to data (ie. data is in an array)
  – The data is ordered

• This means, of course, that we can only do efficient binary search on an array (NOT a linked list)
• In 2006, a bug was found by Google
  – http://googleresearch.blogspot.com/2006/06/

• Where is it? low + high can yield overflow when both are large

• How can we fix it?
  mid = low + ((high-low) / 2)

```c
int _binarySearch(TYPE * data, int size,
                  TYPE val) {
  int low = 0;
  int high = size;
  int mid;
  while (low < high) {
    mid = (low + high) / 2;
    if (EQ(data[mid],val))
      return mid;
    if LT(data[mid],val))
      low = mid + 1;
    else  high = mid;
  }
  return low;
```
How can we maintain sorted order when adding elements?

• **Option 1:** Use a sorting algorithm?

• **Option 2:** Linear search for proper location, then insert?

• **Option 3:** Binary search for proper location, then insert?
Option 1: Sort upon add

• Add 1 item
  – Put it at end: \( O(1) \)
  – sort the items \( O(N\log N) \)

• Overall complexity of Adding 1 item:
  – \( O(1 + N\log N) = O(N\log N) \)
Option 2: Linear search, insert

• Add 1 item
  – Linear search to proper location
  – Insert and shift later items ahead
  – $O(N)$ operations

• Overall complexity of 1 Insert:
  – $O(N)$
Option 3: Binary Search

• Add 1 items
  – _binary Search to find location: O(logN)
  – insert at that location: O(N)

• Total Complexity of N Inserts:
  – O(logN + N) = O(N)

• Faster than linear search option in practice (despite both being O(N))
  – Binary search is a faster way to find insertion point
Which operation is now faster?

Using a dynamic array for an ordered array, which of the following operations is made faster or slower by using a binary search?

- add(element)
- contains(element)
- remove(element)

• Which option (1-3) is best?
  - sort
  - linear search
  - binary search
Your Turn

• Now that we have \_binarySearch, how do the following change?
  – addBag
  – containsBag
  – removeBag

• Complete Worksheet #26