Container Objects and Searching

• Lists, linked lists, trees, and dictionaries are various objects whose principle purpose is to contain values and present them in various ways.

• We’ve principally considered operations that involve retrieving all values and doing something with them.

• But a central activity of many programs and algorithms is finding a value that meets certain criteria in one of these containers.

• Several Python data structures provide methods for finding:

  ```python
  x in aList  # Is x in aList?
  x in aDict  # Is x a key in aDict?
  aDict[x]    # What is V if aDict contains the entry (x, V)?
  "61A" in text  # Does substring ’61A’ appear in string text?
  ```
Sets

• Current versions of Python also have sets, which are intended to behave like mathematical sets.

• Examples:

  A = { 1, 3, 2 }  # Definition by extension
  B = set([1, 3, 5])  # Contents can come from an iterable
  set()  # The empty set
  {}  # The empty dictionary (sorry)
  { x for x in L if x % 2 == 1 }  # Set generator: odd members of L
                                 # Like \{x|x \in L \text{ and } x \text{ is odd}\}
  A | B == { 1, 2, 3, 5 } == A.union(B) # A \cup B
  A & B == { 1, 3 } == A.intersection(B) # A \cap B
  A - B == { 2 } == A.difference(B) == { x for x in A if x not in B }
  A < (A | B) == True # A \subset A \cup B
  3 in A == True  # 3 \in A
  len(A) == 3  # |A| or size of A
Sets are Iterables

- Like other container types, one can iterate over sets.
- Python sets are *unordered*: ordering of iterator results is undefined.

```python
>>> for x in {5000, 3000, 100}: print(x, end=" ")
3000 5000 100
>>> list( {5000, 3000, 100} )
[3000, 5000, 100]
```
Example

How can I test whether a list contains duplicates?

def hasDuplicates(L):
    """Return true iff list L contains duplicated values."""
Implementing Sets: Unordered Lists

- Clearly, lists also contain collections of values, so we could use them to implement sets.
- Must be careful to avoid duplicate elements (important when iterating).
- The algorithm for “member of” \((x \text{ in } S)\) is familiar:

```python
def contains(S, x):
    """True iff list S (considered as a set) contains x.""
    for y in S:
        if x == y:
            return True
    return False
```

- If \(N\) is the length of \(S\), what is the worst-case time bound?
Implementing Sets: Unordered Lists

- Clearly, lists also contain collections of values, so we could use them to implement sets.

- Must be careful to avoid duplicate elements (important when iterating).

- The algorithm for “member of” (x in S) is familiar:

  ```python
  def contains(S, x):
    """True iff list S (considered as a set) contains x.""
    for y in S:
      if x == y:
        return True
    return False
  ```

- If $N$ is the length of $S$, what is the worst-case time bound? Answer: $\Theta(N)$
**Implementing Sets: Insertion/Formation w/ Unordered List**

What's the time required for this? Assume appending to a list takes $O(1)$ time (which is true on average).

```python
def toSet(L):
    """Returns an unordered list containing all values in L without duplicates."""
    result = []
    for x in L:
        if not contains(result, x):
            result.append(x)
    return result
```

Implementing Sets: Insertion/Formation w/ Unordered List

What’s the time required for this? Assume appending to a list takes $O(1)$ time (which is true on average).

```python
def toSet(L):
    """Returns an unordered list containing all values in L without duplicates."""
    result = []
    for x in L:
        if not contains(result, x):
            result.append(x)
    return result

Answer: $\Theta(N^2)$
Implementing Sets: Ordered Lists

- If we keep list sorted (say in ascending order), can use *binary search*: 

```python
def contains(S, x):
    """Returns true if X is in S, a list sorted in ascending order."""
    L, U = 0, len(S)-1
    while L <= U:
        M = (L + U) // 2
        if x == S[M]:
            return True
        elif x < S[M]:
            U = M - 1
        else:
            L = M + 1
    return False
```

- What's the execution time here (if \( N \) is \( \text{len}(S) \))?
Implementing Sets: Ordered Lists

- If we keep list sorted (say in ascending order), can use *binary search*:

```python
def contains(S, x):
    
    """Returns true if X is in S, a list sorted in ascending order.""
    L, U = 0, len(S)-1
    while L <= U:
        M = (L + U) // 2
        if x == S[M]:
            return True
        elif x < S[M]:
            U = M - 1
        else:
            L = M + 1
    return False
```

- What's the execution time here (if \( N \) is `len(S)`)? **Answer**: \( \Theta(\lg N) \)
Implementing Sets: Insertion/Formation w/ Ordered List

What’s the time required for this? Assume appending to a list takes $O(1)$ time (which is true on average).

def toSet(anIterable):
    """Returns an ordered list containing all values in ANITERABLE without duplicates."""
    result = []
    for x in anIterable:
        L, U = 0, len(result)-1
        while L <= U:
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)
Implementing Sets: Insertion/Formation w/ Ordered List

What's the time required for this? Assume appending to a list takes $O(1)$ time (which is true on average).

```python
def toSet(anIterable):
    """Returns an ordered list containing all values in ANITERABLE without duplicates."""
    result = []
    for x in anIterable:
        L, U = 0, len(result)-1
        while L <= U:
            M = (L + U) // 2
            if x == result[M]:
                break
            elif x < result[M]:
                U = M - 1
            else:
                L = M + 1
        if L > U:
            result.insert(L, x)

Answer: $\Theta(N^2)$
Binary Search Trees

Binary Search Property:

- In a *binary tree*, each inner node has two children (called “left” and “right”, typically), but trees are allowed to be *empty* (no label, no children).
- A *binary search tree* (BST) satisfies two other properties:
  - All nodes in left subtree of a node have *smaller* keys.
  - All nodes in right subtree of node have *larger* keys.
- This allows binary search, but in a tree.
Finding

• Example: Searching for 50 and 49 in a BST representing
  \{ 16, 19, 25, 30, 42, 50, 65, 91 \}

```python
def contains(S, x):
    """Returns true iff BST S contains x.""
    if S == BinTree.empty:
        return False
    if S.label == x:
        return True
    elif S.label < x:
        return contains(S.right, x)
    else:
        return contains(S.left, x)
```

- Dashed boxes show which node labels we look at.
- Number looked at proportional to height of tree.
- What is worst-case time (for a general tree with \( N \) nodes)?
- If tree is “bushy,” what is worst-case time?
Finding

• Example: Searching for 50 and 49 in a BST representing
  \{ 16, 19, 25, 30, 42, 50, 65, 91 \}

```
def contains(S, x):
    """Returns true iff BST S contains x."""
    if S == BinTree.empty:
        return False
    if S.label == x:
        return True
    elif S.label < x:
        return contains(S.right, x)
    else:
        return contains(S.left, x)
```

- Dashed boxes show which node labels we look at.
- Number looked at proportional to height of tree.
- What is worst-case time (for a general tree with \(N\) nodes)? \textbf{Answer:} \(\Theta(N)\)
- If tree is “bushy,” what is worst-case time?

Finding

- Example: Searching for 50 and 49 in a BST representing
  \{ 16, 19, 25, 30, 42, 50, 65, 91 \}

```python
def contains(S, x):
    """Returns true iff BST S contains x."""
    if S == BinTree.empty:
        return False
    if S.label == x:
        return True
    elif S.label < x:
        return contains(S.right, x)
    else:
        return contains(S.left, x)
```

- Dashed boxes show which node labels we look at.
- Number looked at proportional to height of tree.
- What is worst-case time (for a general tree with \( N \) nodes)? **Answer:** \( \Theta(N) \)
- If tree is "bushy," what is worst-case time? **Answer:** \( \Theta(\lg N) \)
def add(S, x):
    """Add X to binary search tree S destructively, if not already present, returning new tree."""
    if S == BinTree.empty:
        return BinTree(x)
    elif S.label < x:
        S.right = add(S.right, x)
    else:
        S.left = add(S.left, x)
    return S

• Starred edges are set (to themselves, unless initially null).

• Again, time proportional to height.
What Does Python Do?

- Binary trees are just a special case of this algorithm from Lecture #19:
  ```python
def tree_find(T, disc):
    p = disc(T.label)
    if p == -1:
        return T.label
    elif T.is_leaf():
        return None
    else:
        return tree_find(T.children[p], disc)
```
  where the discrimination function (\texttt{disc}) returns either -1 (when the label is the target), 0 (for left child), or 1.

- In effect, for its sets (and dictionaries), Python uses another specialization of this same algorithm, where \texttt{disc} can return values in an arbitrary range, and the tree is always height 1.

- The discrimination function in this case is called a \textit{hashing function}.
Hashing

- Example: the previous set of integers:
  \[
  \{ 16, 19, 25, 30, 42, 50, 65, 91 \}
  \]

  where the hashing function returns the value of the last digit.

- The tree labels on the leaves can be simple unordered lists of values, each sharing the same hashed value (their last digit in this case).

- As long as these lists stay small, look-up time is short. In fact, if there is a constant bound on list size, look-up time is $\Theta(1)$.

- When lists get too long, just increase the number of children at the root.
More Details

• To allow this to work, must define a hash function for your data.

• The Python way is to add a method `__hash__`, which is expected to return an integer such that the value returned for two objects that are considered equal (`==`) are equal.

• Python chooses the number of children (which are called `buckets`) of the top node depending on the current number of items in the set or dictionary represented.

• It then computes a discriminating value between between 0 and $N$, the number of children, by some process such as taking the value of `__hash__` modulo $N$. 