Topic 8
Lecture 8b
Sets and Selection

CSCI 240

Data Structures and Algorithms

Prof. Dominick Atanasio

The Set ADT

- A set is a collection of distinct objects.
  - There are no duplicate elements in a set
  - There is no explicit notion of keys or even an order.
- If the elements in a set are comparable, then we can maintain sets to be ordered.

Interface of the Set ADT

- The fundamental functions of the set ADT for a set S are the following:
- insert(e): Insert the element e into S and return an iterator referring to its location; if the element already exists the operation is ignored.
  - find(e): If S contains e, return an iterator p referring to this entry, else return end.
  - erase(e): Remove the element e from S.
  - begin(): Return an iterator to the beginning of S.
  - end(): Return an iterator to an imaginary position just beyond the end of S.

STL Implementation

- The C++ Standard Template Library provides a class set that contains all of these functions.
- It actually implements an ordered set and supports the following additional operations as well.
  - lower bound(e): Return an iterator to the largest element less than or equal to e.
  - upper bound(e): Return an iterator to the smallest element greater than or equal to e.
  - equal range(e): Return an iterator range of elements that are equal to e.

Mergeable Sets

- A further extension of the ordered set ADT that allows for operations between pairs of sets.
- To discuss mergeable Sets we will look at the following three set operations:
  1. Union
  2. Intersection
  3. Subtraction
- Assume these operations are performed on two sets: A and B, and a resultant set C such that:

$$C = A \cup B = \{\forall x \in C: x \in A \text{ or } x \in B\}$$

$$C = A \cap B = \{\forall x \in C: x \in A \text{ and } x \in B\}$$

$$C = A - B = \{\forall x \in C: x \in A \text{ and } x \notin B\}$$

Implementation

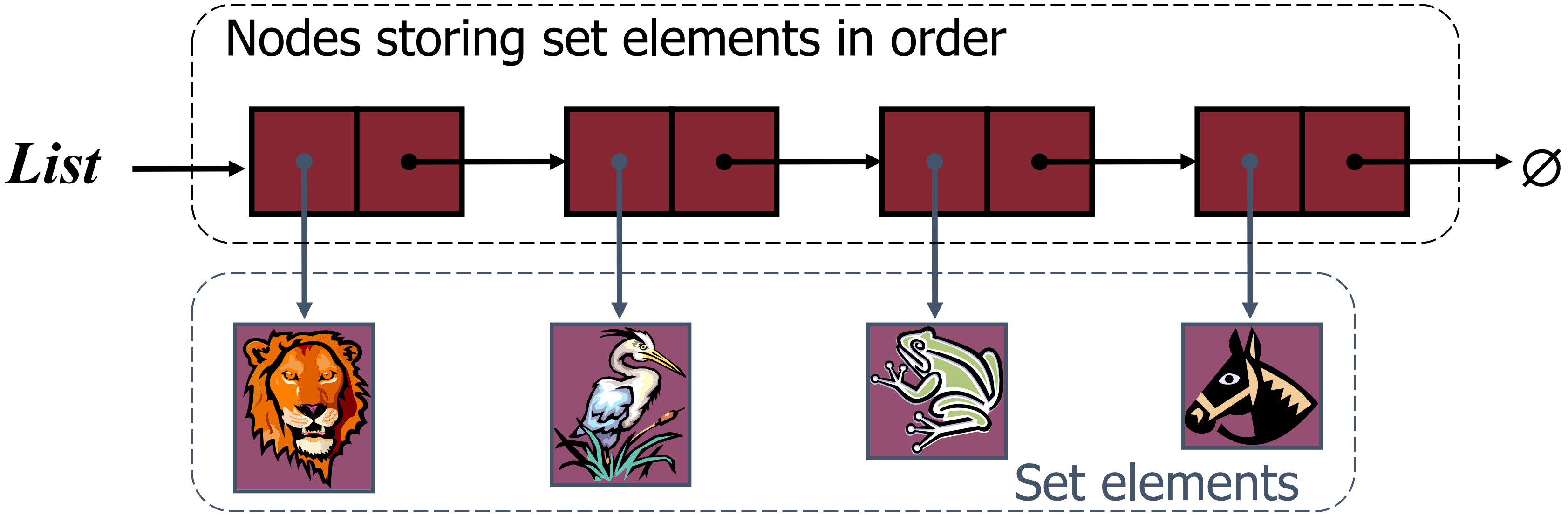
- One of the simplest ways of implementing a set is to store its elements in an ordered sequence.
- This implementation is included in several software libraries for generic data structures
- Cs consider implementing the set ADT with an ordered sequence
- Any consistent total order relation among the elements of the set can be used, provided the same order is used for all the sets.
- The C++ STL implements its ordered set as a BST

Set Operations

- By specializing the auxiliary functions this generic merge algorithm can be used to perform basic set operations:
  - union
  - intersection
  - subtraction
- The generic merge algorithm iteratively examines and compares the current elements, a and b of the input sequence A and B, respectively, and finds out whether a < b, a = b, or a > b.
- The running time of an operation on sets A and B should be at most $O(n_A + n_B)$

Storing a Set in a List

- We can implement a set with a list (can use a skiplist)
- Elements are stored sorted according to some canonical ordering
- The space used is O(n)



Generic Merging

- Algorithm for the generic merge

```
MERGE(List& A, List& B, out List& C)
START
   Iterator ia = start of A
   Iterator ib = start of B
   WHILE(ia not at end of A AND ib not at end of B)
      if(ia < ib) then fromA(ia++, C)
      else if(ia == ib) then fromBoth(ia++, ib++, C)
      else fromB(ib++, C)
   END WHILE
   WHILE(ia != end of A) fromA(ia++, C) END WHILE
   WHILE(ib != end of B) fromB(ib++, C) END WHILE
END MERGE
```

Specialized Functions: UnionMerge

```
fromA(const Iterator& a, List& C)
     C.addBack(a) // add a

fromBoth(const Iterator& a, const Iterator& b, List& C)
     C.addBack(a) // add a only since a == b

fromB(const Iterator& b, List& C)
     C.addBack(b) // add b
```

Specialized Functions: IntersectMerge

```
fromA(const Iterator& a, List& C)
     return // do nothing

fromBoth(const Iterator& a, const Iterator& b, List& C)
     C.addBack(a) // add a only since a == b

fromB(const Iterator& b, List& C)
     return // do nothing
```

Specialized Functions: SubtractMerge

```
fromA(const Iterator& a, List& C)
     C.addBack(a) // add a

fromBoth(const Iterator& a, const Iterator& b, List& C)
     return // do nothing

fromB(const Iterator& b, List& C)
     return // do nothing
```

Using Generic Merge for Set Operations

- Any of the set operations can be implemented using a generic merge
- For example:
  - For intersection: only copy elements that are duplicated in both list
  - For union: copy every element from both lists except for the duplicates
- All methods run in linear time

Selection

Selection

- There are several applications in which we are interested in identifying a single element in terms of its rank relative to an ordering of the entire set.
- A trivial requirement is to identify the minimum and maximum elements.
- We may also be interested in, say, identifying the median element.
  - The median element is the element such that half of the other elements are smaller and the other half are larger

The Selection Problem

- The selection problem is the general order-statistic problem of selecting the k^{th} smallest element from an unsorted collection of n comparable elements.
- The collection could be sorted and then indexed into the sorted sequence at $k - 1$.
  - Using the best comparison-based sorting algorithms, this approach would take O(*nlogn*) time.
  - This may take more time than is necessary to solve the problem.
  - What if it can be done in O(*n*) time? It can be!

The Prune-and-Search Algorithm

- AKA reduce-and-conquer
- We solve a given problem that is defined on a collection of *n* objects by pruning away a fraction of the n objects with each recursive call which then solves the smaller problem.
- When the problem is reduced to one defined on a constant-sized collection of objects, we then solve the problem using some brute-force method
- Of course, we can avoid using recursion, in which case we simply iterate the prune-and-search reduction step until we can apply a brute-force method.

Randomized Quick-select

- A simple and practical method, called randomized quick-select, for finding the k^{th} smallest element in an unordered sequence of n elements on which a total order relation is defined
- We've seen this in the Quicksort algorithm
- Randomized quick-select runs in $O(n)$ expected time
- Suppose we are given an unsorted sequence S of n comparable elements together with an integer $k \in [1,n]$.
- We pick an element x from S at random
- We then use this as a "pivot" to subdivide S into three subsequences L, E, and G, which store the elements of S less than x, equal to x, and greater than x, respectively.
- Based on the value of k, we determine which of these sets to recur on.

Randomized Quick-select Algorithm

Algorithm quickSelect(S,k):

Input: Sequence S of n comparable elements, and an integer $k \in [1,n]$

Output: The kth smallest element of S

if $n = 1$ **then**

  return the (first) element of S.

pick a random (pivot) element x of S and divide S into three sequences:

- L, storing the elements in S less than x
- E, storing the elements in S equal to x
- G, storing the elements in S greater than x.

if $k \leq |L|$ **then**

  quickSelect(L,k)

else if $k \leq |L| + |E|$ **then**

  return x $\{$each element in E is equal to $x\}$

else

  quickSelect$(G, k - |L| - |E|)$ $\{$note the new selection parameter$\}$