Chapter 4

ADT Sorted List

Sorted List: Logical Level

- Only change from unsorted list is guaranteeing list elements are sorted
- Order is determined by ItemType's CompareTo method
- PutItem and DeleteItem pre- and postconditions assert list is sorted and remains sorted

Sorted List: Application Level

- Nothing has changed for the user; list interface is exactly the same
- GetNextItem will return the next item in key order

Sorted List: Implementation Level

- Few changes are needed to implement the sorted list
- PutItem, DeleteItem: Ensure list remains sorted
- GetItem can be improved
- First attempt: Will implement an array-based list

PutItem Implementation

- Algorithm is simple:
  - Find the place where new element should go
  - Create space for the new element
  - Insert the element in the space
- A linear search is sufficient for finding the position
- Create space by moving all subsequent elements down one space

PutItem Implementation (cont.)

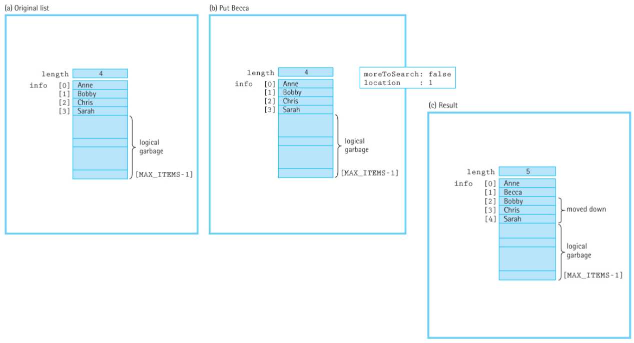

Figure 4.1 Putting an item into a sorted list (a) Original list (b) Put Becca (c) Result

DeleteItem Implementation

- We assume the item to be deleted is in the list
- Again, simple linear search to find the item
- Once the item is found
  - Move every subsequent element up one space (overwriting the element to be deleted)
  - Decrement length by one

GetItem Implementation

- Using a sorted list allows us to improve on Unsorted List's linear search
- Simple optimization: Stop searching the list when we encounter an item greater than what we're searching for
  - Example: List is "Alice," "Bob," "Diane," "Ed" and we're searching for "Chris"
  - Search can stop once we reach "Diane" because "Chris" must come before "Diane"

GetItem: Binary Search

- **Binary Search:** An O($\log_2 N$) search algorithm where the search area is halved on every iteration
  - Stops when item is found or when there's nothing left to search (item is not in list)
- Requires a sorted list and random access to list elements (i.e., an array-based list)
- Linear search can be faster for smaller *N* (<20)

Binary Search Walkthrough

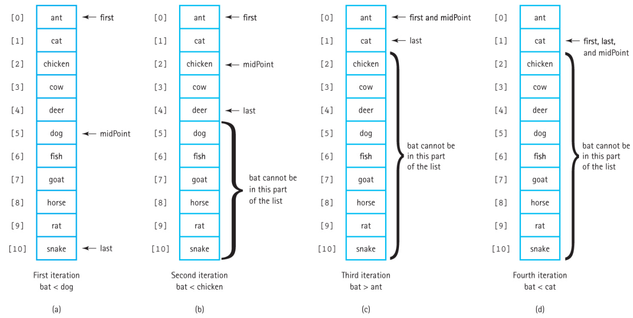

Figure 4.4 Trace of the binary search algorithm (a) First iteration bat < dog (b) Second iteration bat < chicken (c) Third iteration bat > ant (d) Fourth iteration bat < cat)

Dynamically Allocated Arrays

- Dynamically allocated arrays allow clients to decide how many elements to store at run time
- Use a pointer to an array of ItemType on the heap
- Requires very few changes to implement

Dynamically Allocated Arrays (cont.)

- Parameterized constructor: Allows user to specify max number of items
- Default constructor: Allocated 500 items; used for arrays of lists
- Destructor: Cleans up the memory on the heap when the rest of the list is removed
- IsFull: We store the max list size instead of using a constant: `length == maxList`

Dynamically Allocated Arrays (cont.)

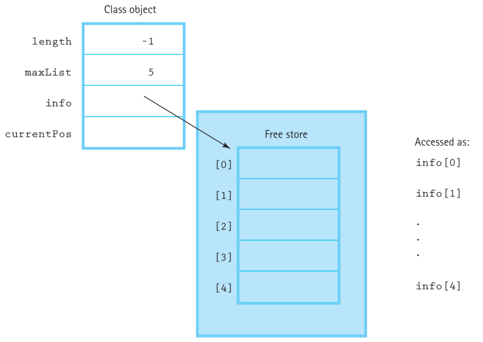

Figure 4.5 A list in which the array is in dynamic storage

Linked List-Based Sorted List

- Very similar to changing array-based Unsorted List to linked list-based
- GetItem changes since searching the list is different
- PutItem and DeleteItem must handle changing links between nodes

Linked List Implementation: GetItem

- The same simple optimization is usable: Stop once we reach an item that comes after the item we're searching for
  - If CompareTo returns Equal, item has been found
  - If CompareTo returns Less, item is not in the list
- Cannot use Binary search here. Why?
  - Binary search requires being able to randomly access elements of the list
  - Linked lists can only access directly linked nodes

Linked List Implementation: PutItem

- The algorithm starts the same: Search through the list to find the location for the new item
  - Example: List contains "Alex," "Chris," "John," and "Kit," and we want to insert "Becca"
  - Search stops at "Chris" because "Becca" < "Chris"
  - Prepare a new node for "Becca," which points to "Chris" as the next node
  - We need "Alex" to point to "Becca," but since we don't have a pointer to "Alex," we can't modify it!

Linked List Implementation: PutItem (cont.)

- Possible solution: Always look ahead one node
  - That is, compare against `(location->next)->info`
  - But this will cause an exception when we reach the end of the list
- Solution: Use a second pointer that points to the previous node
  - Will be NULL if inserting at the beginning of the list, which is a special case

Linked List Implementation: PutItem (cont.)

- We update both pointers at the same time, creating an "inchworm effect"

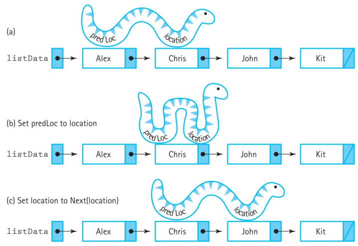

Figure 4.9 The inchworm effect (a) (b) Set predLoc to location (c) Set location to Next(location)

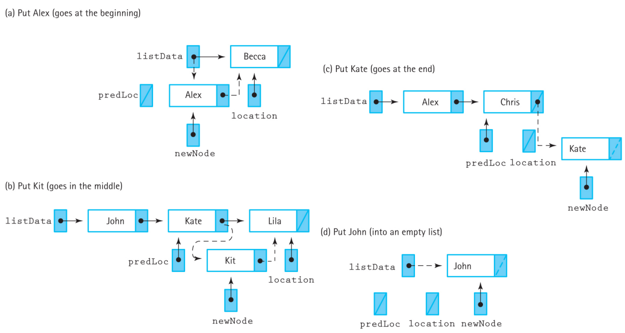

Figure 4.10 Four insertion cases (a) Put Alex (goes at the beginning) (b) Put Kit (goes in the middle) (c Put Kate (goes at the end) (d) Put John (into an empty list)

Linked List: DeleteItem

- DeleteItem doesn't change much, but it must handle updating pointers
- Two options:
  - Use the same algorithm as the unsorted list: compare against `(location->next)->info` to find the item to delete
  - Mirror the PutItem algorithm so it removes the node using two location pointers
- As with PutItem, there are four cases to consider

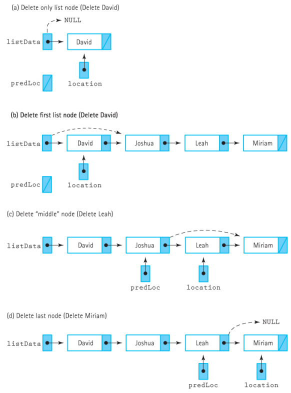

Figure 4.11 Deleting from a linked list: (a) Delete only list node (Delete David) (b) Delete first list node (Delete Radio) (c) Delete "middle" node (Delete Leah) (d) Delete last node (Delete Miriam)

Comparing Sorted List Implementations

- GetItem: Array-based can use binary search, which is O($\log_2 N$), but linked list-based is stuck with sequential O(N) search
- PutItem: Array-based requires searching plus moving all subsequent items, while linked list-based only needs to search; both are O(N), but linked list-based does less work overall
- DeleteItem: Same as PutItem; both are O(N) and therefore roughly equivalent

Sorted List Implementations

Table 4.3 Big-O Comparison of Sorted List Operations

	Array Implementation	Linked Implementation
class constructor	O(1)	O(1)
MakeEmpty	**O(1)**	**O(*N*)**
IsFull	O(1)	O(1)
GetLength	O(1)	O(1)
ResetList	O(1)	O(1)
GetNextItem	O(1)	O(1)
GetItem	O(*N*)*	O(*N*)
PutItem		
Find	O(*N*)*	O(*N*)
Put	**O(*N*)**	**O(1)**
Combined	O(*N*)	O(*N*)
DeleteItem		
Find	O(*N*)*	O(*N*)
Delete	**O(*N*)**	**O(1)**
Combined	O(*N*)	O(*N*)

*O($\log_2 N$) if a binary search is used.

Table 4.3 Big-O Comparison of Sorted List Operations

Sorted List vs. Unsorted List

- The main difference is PutItem
  - Unsorted List is O(1) because it doesn't care about order
  - Sorted List is O(N) because it must search the list to find where the item should go
- GetItem in a Sorted List can be more efficient than an Unsorted List if binary search is used

Bounded and Unbounded ADTs

- **Bounded ADT:** There is a logical limit on the number of items in the structure
- **Unbounded ADT:** There is no logical limit on the number of items in the structure
- Are our lists bounded or unbounded?

Bounded or Unbounded Lists

- The array-based list is bounded and the linked list-based list is unbounded, but this is an *implementation difference*
- The dynamically allocated array-based list uses arrays but can be unbounded by allocating more memory as needed
- Clients only know an ADT is bounded or unbounded if the documentation tells them so

Object-Oriented Design Methodology

Four main steps to object-oriented design:

- Brainstorming
- Filtering
- Scenarios
- Responsibility Algorithms

Brainstorming

- A group problem-solving technique where all members of the group spontaneously contribute ideas
  - All ideas are potentially good ideas
  - Think fast and furiously, and ponder later
  - Give every voice a turn
  - A little humor can be a powerful force
- The goal is to produce a list of candidate classes for solving the problem at hand

Filtering

- Take the list of classes from the Brainstorming step and determine which classes are core to solving the problem
- You may find some classes are duplicates, or others could be combined into a single class
- Once complete, CRC cards should be written for the remaining classes

Scenarios

- **Scenario:** A sequence of steps that describes an interaction between a client and a program
  - A set of related scenarios is a **use case**
- Goal of this step is to assign responsibilities to each class
  - Responsibility: A task handled by a class; something a class should know or be able to do
- Scenarios help us decide the tasks in a program and which classes handle each task

Responsibility Algorithms

- This step involves writing the algorithms for each responsibility
- Knowledge responsibilities usually just return data encapsulated by the class
- The top-down design approach is usually sufficient for implementing action responsibilities