Data Structures and Algorithms

Spring 2008-2009

Outline

Introduction

- An ADT is a data entity with operations associated with that entity
- The ADT does not specify how the operations are implemented
- Some of the ADTs we will encounter:
  - Lists
  - Trees
  - Graphs
- Implementation of an operation may change but this must *not* affect a user of the ADT

Outline

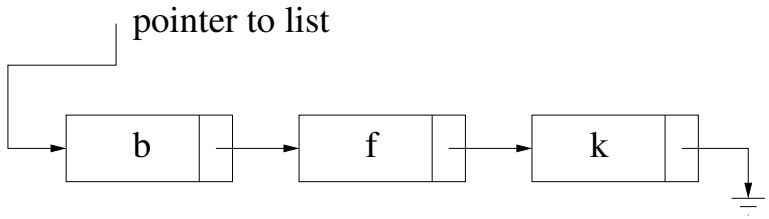
Main Points

- In dynamic situations records may be inserted and deleted with great frequency
- Storing records in a sorted array gives poor performance:
  - ✓ while an element can be found using binary search in $O(\log n)$-time
  - ✗ to insert or delete an element, time taken is on average $\log n + \frac{n}{2} = O(n)$ if the array is sorted

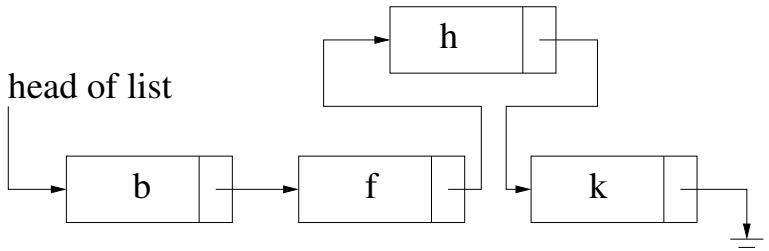
Main Points (contd.)

To store data items on a linked list

- we store each item in a cell, and
- each cell contains a pointer to the next cell



- using an *unsorted* list ADT we can achieve $O(1)$-time for insertions at the expense of $O(n)$-time for deletions and searches
- however, for *sorted* lists, insertion- or deletion- or search-time is on avg. $\frac{n}{2} = O(n)$



Some Issues with Lists

How do we handle:

- insertions before the first element
- deletion of first element

Weiss' linked list implementation:

- Header (declaration) file: `LinkedList.h`
  (Note `ListItr` class that akwardly implements *iterators*)
- Source (definition) file: `LinkedList.cc`
- Test harness file: `TestLinkedList.cc`

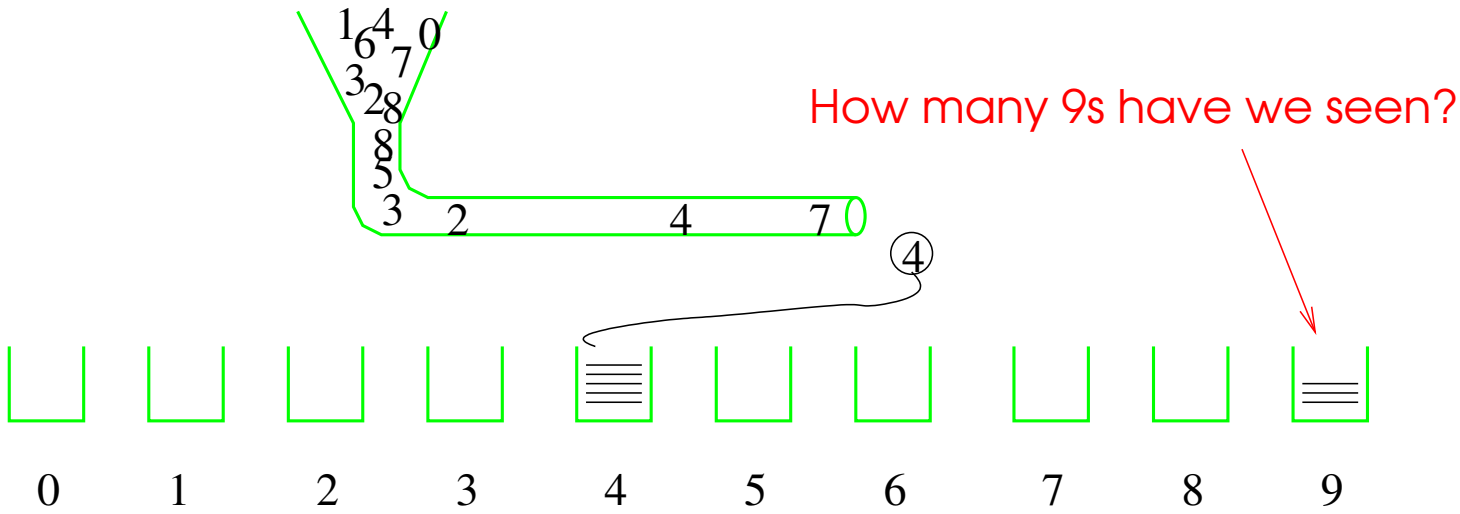
Outline

- Polynomial Arithmetic
- Lists of Lists: Student Database System
- Radix Sort

Bucket Sort

- With n integers in the range $0 \ldots m-1$ we can sort them as follows:
  - create an array, `arr`, of m buckets and initialize each bucket to 0
  - for each integer, i, of the n integers, bump `arr[`i`]`
  - now run through each index, j, of `arr`, printing out j `arr[`j`]` times
- running time of this algorithm is $O(m+n) = O(\max(m,n))$
- problems: storage and running time is a function of magnitude of integers

Sort 1001 integers in range 0 – 9



Radix Sort

- To sort integers (or strings of letters) in range $0 \ldots b^p - 1$ it is infeasible to allocate b^p buckets (as we did with single digits)
- Instead, use b buckets and make p passes over the data, sorting by the position one to the left each time
- To sort integers, $b = 10$; to sort alphabetic strings, $b = 27$
- During the ith iteration, each bucket will be a linked list of all the items with identical radix in the ith least significant position
- At the end of the iteration the buckets are emptied
  - from left to right (lesser elements before greater), and
  - from bottom of bucket to top
- This suggests a *list* data structure for storing within each bucket

Radix Sort (contd.)

- Running time of radix sort is $O(p(b+n))$ since the algorithm does p bucket sorts
- Can sort any set of integers representable in 32 bits in four passes if we use 2^8 buckets where each bucket will hold integers that have the same 8-bit block in one of the four radix positions
- Why does radix sort work? What is the ordering in a bucket after a pass of the algorithm?

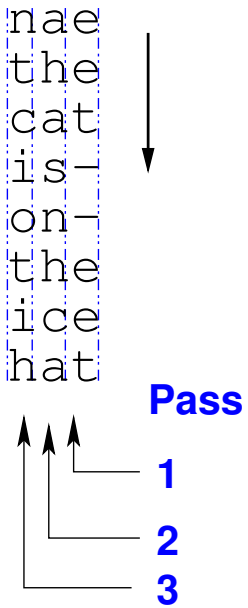
Radix Sort (contd.)

Example Sort the words of the phrase:

`nae the cat is on the ice hat`

Preprocessing step: since longest word has 3 letters, pad out all other words with dashes to be 3 “letters” long; a ’–’ precedes an ’a’ in the alphabet

Radix Sort (contd.)



After pass 1:

'a'	'c'	'h'
on- is-	ice the the nae	hat cat

Before pass 2: is- on- nae the the ice cat hat

After pass 2:

'a'	'c'	'h'	'n'	's'
hat cat nae	ice	the the	on-	is-

Before pass 3: nae cat hat ice the the on- is-

After pass 3:

'c'	'h'	'i'	'n'	'o'	't'
cat	hat	is- ice	nae	on-	the the

Final output: cat hat ice is- nae on- the the