Advanced Algorithmics (6EAP)

Linear structures, sorting, searching, etc

Jaak Vilo
2012 Spring

Physical ordered list ~ array

- Memory /address/
  - Garbage collection

- Files (character/byte list/lines in text,...)

- Disk
  - Disk fragmentation

Lists: Array

L = int[MAX_SIZE]
L[2]=7

Multiple lists, 2-D-arrays, etc...

L[2]=7
**Linear Lists**

- Operations which one may want to perform on a linear list of $n$ elements include:
  - gain access to the $k$th element of the list to examine and/or change the contents
  - insert a new element before or after the $k$th element
  - delete the $k$th element of the list

**Abstract Data Type (ADT)**

- High-level definition of data types
- An ADT specifies
  - A collection of data
  - A set of operations on the data or subsets of the data
- ADT does not specify how the operations should be implemented
- Examples
  - vector, list, stack, queue, deque, priority queue, table (map), associative array, set, graph, digraph

**Abstract data types:**

- Dictionary (key,value)
- Stack (LIFO)
- Queue (FIFO)
- Queue (double-ended)
- Priority queue (fetch highest-value object)
  - ...

**ADT**

- A datatype is a set of values and an associated set of operations
- A datatype is abstract if it is completely described by its set of operations regardless of its implementation
- This means that it is possible to change the implementation of the datatype without changing its use
- The datatype is thus described by a set of procedures
- These operations are the only thing that a user of the abstraction can assume

**Dictionary**

- Container of key-element (k,e) pairs
- Required operations:
  - insert(k,e),
  - remove(k),
  - find(k),
  - isEmpty()
- May also support (when an order is provided):
  - closestKeyBefore(k),
  - closestElemAfter(k)
- Note: No duplicate keys

**Abstract data types**

- Container
- Deque
- Map/Associative array/Dictionary
- Multimap
- Multiset
- Priority queue
- Queue
- Set
- Stack
- String
- Tree
- Graph
- Hash
Some data structures for Dictionary ADT

- Unordered
  - Array
  - Sequence/list
- Ordered
  - Array
  - Sequence (Skip Lists)
  - Binary Search Tree (BST)
  - AVL trees, red-black trees
  - 2-4 Trees
  - B-Trees
- Valued
  - Hash Tables
  - Extendible Hashing

Primitive & composite types

- **Primitive types**
  - Boolean (for boolean values True/False)
  - Char (for character values)
  - Int (for integral or fixed-precision values)
  - Float (for storing real number values)
  - Double (larger size of type float)
  - String (for string of characters)
  - Enumerated type

- **Composite types**
  - Array
  - Record (also called tuple or struct)
  - Union
  - Tagged union (also called a variant, variant record, discriminated union, or disjoint union)
  - Plain old data structure

Linear data structures

**Arrays**
- Array
- Bidirectional map
- Bit array
- Bit field
- Bitboard
- Bitmap
- Circular buffers
- Control table
- Image
- Dynamic array
  - Gap buffer
  - Hashed array
  - Heuristics
  - Lookup table
  - Matrix
  - Parallel array
  - Sparse array
  - Sparse matrix
  - UHFs vector
  - Variable-length array

**Lists**
- Doubly linked list
- Linked list
- Self-organizing list
- Skip list
- Unrolled linked list
- VLST
- XOR linked list
- Zipper
- Doubly connected edge list

Hashes, Graphs, Other

- Hashes
  - Bloom filter
  - Distributed hash table
  - Hash array mapped trie
  - Hash list
  - Hash table
  - Hash trie
- Koord
- Prefix hash tree

- Graphs
  - Adjacency list
  - Adjacency matrix
  - Graph-structured stack
  - Scene graph
  - Binary decision diagram
  - Zero-suppressed decision diagram
  - And-inverter graph
  - Directed graph
  - Directed acyclic graph
  - Propositional directed acyclic graph
  - Multigraph
  - Hypergraph

- Other
  - Lightmap
  - Winged edge
  - Quad-edge
  - Routing table
  - Symbol table

Trees

- Binary tree
  - AVL trees
  - Balanced BST
  - Binomial tree
  - B-Tree
  - D-Tree
  - Red-black trees
  - Splay tree
  - T-Tree

- Heap
  - 2-3 tree
  - 2-3 heap
  - 2-3-4 tree
  - 2-3-4 heap
  - AF-heap
  - Binary heap
  - Max heap
  - Min heap
  - Min/Max heap
  - Priority queue
  - Ternary heap

- Heaps
  - Min/Max heap
  - Priority queue

- Tries
  - Generalized trie
  - Tries
  - Ternary search tree
  - AF-trie

- Other
  - kmd-tree
  - Hash table
  - Sorting
  - Point data structure
  - Union
  - Linear array
  - Hash table
  - Array
  - Insertion sort
  - Quick sort
  - Merge sort
  - Min/Max heap
  - Heapify
  - Search

Lists: Array

```
0 1
size
3 6 7 5 2
MAX_SIZE-1
```

```
0 1
size
3 6 7 8 5 2
Insert 8 after L[2]
```

```
0 1
size
3 6 7 8 5 2
Delete last
```
Lists: Array

- Insert O(n)
- Delete O(n)
- Access i O(1)
- Insert to end O(1)
- Delete from end O(1)
- Search O(n)

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

Insert 8 after L[2]

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

Delete last

Linear Lists

- Other operations on a linear list may include:
  - determine the number of elements
  - search the list
  - sort a list
  - combine two or more linear lists
  - split a linear list into two or more lists
  - make a copy of a list

Stack

- push(x) -- add to end (add to top)
- pop() -- fetch from end (top)

- O(1) in all reasonable cases

- LIFO – Last In, First Out

Linked lists

Singly linked

Doubly linked

Linked lists: add

Linked lists: delete

(+ garbage collection?)
Operations

- Array indexed from 0 to \( n - 1 \):

<table>
<thead>
<tr>
<th>access/change the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

- Singly-linked list with head and tail pointers

<table>
<thead>
<tr>
<th>access/change the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert before or after the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delete the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

- Doubly-linked list:

<table>
<thead>
<tr>
<th>access/change the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert before or after the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delete the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

Under the assumption we have a pointer to the \( k \)th node, \( O(1) \) otherwise.

Improving Efficiency

<table>
<thead>
<tr>
<th>access/change the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert before or after the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delete the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Doubly-linked list:</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert before or after the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delete the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Singly-linked list with head and tail pointers</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert before or after the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delete the ( k )th element</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Doubly linked list</th>
<th>( k = 1 )</th>
<th>( 1 &lt; k &lt; n )</th>
<th>( k = n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

Introducing linked lists

- Consider the following struct definition

```c
struct node {
    string word;
    int num;
    node *next;  // pointer for the next node
};
node *p = new node;
```

Improving Run-Time Efficiency

- We can improve the run-time efficiency of a linked list by using a doubly-linked list:

- Improvements at operations requiring access to the previous node.
- Increases memory requirements...

Introduction to linked lists: inserting a node

- node *p;
- p = new node;
- p->num = 5;
- p->word = "Ali";
- p->next = NULL

![Linked List Diagram]
16.2.2012

**Introduction to linked lists: adding a new node**

- How can you add another node that is pointed by `p->link`?
- `node *p;`
- `p = new node;`
- `p->num = 5;`
- `p->word = "Ali";`
- `p->next = NULL;`
- `node *q;`

```
5 Ali
  num  word  link
```

**Introduction to linked lists**

```
node *p, *q;
p = new node;
p->num = 5;
p->word = "Ali";
p->next = NULL;
q = new node;
q->num = 8;
q->word = "Veli";
```

```
5 Ali   8 Veli
  num  word  next  num  word  next
```

**Pointers in C/C++**

```
p = new node; delete p;
p = new node[20];
p = malloc( sizeof( node ) ); free p;
p = malloc( sizeof( node ) * 20 );
(p+10)->next = NULL; /* 11th elements */
```

**Book-keeping**

- `malloc, new – “remember” what has been created free(p), delete` (C/C++)
- When you need many small areas to be allocated, reserve a big chunk (array) and maintain your own set of free objects
- Elements of array of objects can be pointed by the pointer to an object.
Object

• Object = new object_type;

• Equals to creating a new object with necessary size of allocated memory (delete can free it)

Some links


• Pointer basics: [http://cslibrary.stanford.edu/106/](http://cslibrary.stanford.edu/106/)


Alternative: *arrays and integers*

• If you want to test pointers and linked list etc. data structures, but do not have pointers familiar (yet)

• Use arrays and indexes to array elements instead...

Replacing pointers with array index

Maintaining list of free objects

Multiple lists, single free list
Hack: allocate more arrays ...

use integer division and mod

\[ \text{AA} \{\frac{i-1}{7}\} \rightarrow \{ (i-1) \mod 7 \} \]

\[ \text{LIST}(10) = \text{AA} \{1\} \{2\} \]

\[ \text{LIST}(19) = \text{AA} \{2\} \{5\} \]

Queue (FIFO)

Queue (basic idea, does not contain all controls!)

Circular buffer

A circular buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

Circular Queue
### Queue
- `enqueue(x)` - add to end
- `dequeue()` - fetch from beginning
- **FIFO** – First In First Out
- O(1) in all reasonable cases 😊

### Stack
- `push(x)` -- add to end (add to top)
- `pop()` -- fetch from end (top)
- O(1) in all reasonable cases 😊
- **LIFO** – Last In, First Out

### Stack based languages
- **Implement a postfix calculator**
  - Reverse Polish notation
  - \( 5\ 4\ 3\ *\ 2\ -\ + \) \(\Rightarrow\ \ 5+(4*3)-2\)
- Very simple to **parse** and interpret
- **FORTH, Postscript** are stack-based languages

### Array based stack
- **How to know how big a stack shall be?**
  - When full, allocate bigger table dynamically, and copy all previous values there
  - O(n) ?

### Characteristics of Array based stack
- When full, create 2x bigger table, copy previous n elements:
  - After every \(2^k\) insertions, perform O(n) copy
  - O(n) individual insertions +
  - \(n/2 + n/4 + n/8 \ldots\) copy-ing
  - Total: O(n) effort!

### Characteristics of Stack based languages
- when \(n=32\) -> 33 \(\) (copy 32, insert 1)
- delete: 33->32
  - should you delete immediately?
  - Delete only when becomes less than 1/4th full
  - Have to delete at least \(n/2\) to decrease
  - Have to add at least \(n\) to increase size
  - Most operations, O(1) effort
  - But few operations take O(n) to copy
  - For any \(m\) operations, O(m) time
Amortized analysis

- Analyze the time complexity over the entire "lifespan" of the algorithm
- Some operations that cost more will be "covered" by many other operations taking less

Lists and dictionary ADT...

- How to maintain a dictionary using (linked) lists?
  - Is k in D ?
    - go through all elements d of D, test if d==k  O(n)
    - if sorted: d= first(D); while( d<=k) d=next(D);
    - on average \( n/2 \) tests...
- Add(k,D) \( \Rightarrow \) insert(k,D) = \( O(1) \) or \( O(n) \) – test for uniqueness

Array based sorted list

- is d in D ?
- Binary search in D

Work performed

- \( x \leftrightarrow A[18] \)? <
- \( x \leftrightarrow A[9] \)? >
- \( x \leftrightarrow A[13] \)? ==
- \( O(\log n) \)
16.2.2012

**Sorting**

- given a list, arrange values so that
  \( L[1] \leq L[2] \leq \ldots \leq L[n] \)
- \( n \) elements \( \Rightarrow \) \( n! \) possible orderings
- One test \( L[i] \leq L[j] \) can divide \( n! \) to 2
  - Make a binary tree and calculate the depth
  - \( \log( n! ) = \Omega( n \log n ) \)
- Hence, lower bound for sorting is \( \Omega( n \log n ) \)
  - using comparisons...

**Proof:** \( \log(n!) = \Omega( n \log n ) \)

- \( \log( n! ) = \log n + \log (n-1) + \log(n-2) + \ldots + \log(1) \)
  
  \[ \geq n/2 * \log( n/2 ) \]
  
  \[ = \Omega( n \log n ) \]

---

**Decision-tree example**

Sort \((a_1, a_2, a_3) = (9, 4, 6):\)

Each leaf contains a permutation \((\pi(1), \pi(2), \ldots, \pi(n))\) to indicate that the ordering \(a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}\) has been established.

**Decision tree model**

- \( n! \) orderings (leaves)
- Height of such tree?

\[
\lceil \log_2 n! \rceil \geq \log_2 n!
\]

\[
\geq \sum_{i=1}^{n} \log_2 i
\]

\[
\geq \sum_{i=1}^{n/2} \log_2 n/2
\]

\[
= n/2 \log_2 n/2 = \Omega(n \log n).
\]

- \( \log( n! ) = \log(n)+\log(n-1)+\ldots+\log(1) \)
  
  a) \( \leq n \log(n) \)
  
  b) \( \geq n/2 * \log( n/2 ) = n/2 \log n - n/2 \)
The divide-and-conquer design paradigm

1. Divide the problem (instance) into subproblems.
2. Conquer the subproblems by solving them recursively.
3. Combine subproblem solutions.

Merge sort

\[
\text{Merge-Sort}(A, p, r) \\
\text{if } p < r \\
\quad \text{then } q = (p+r)/2 \quad \text{// floor} \\
\quad \text{Merge-Sort}(A, p, q) \\
\quad \text{Merge-Sort}(A, q+1, r) \\
\quad \text{Merge}(A, p, q, r)
\]

It was invented by John von Neumann in 1945.

Example

• Applying the merge sort algorithm:

Merge of two lists: \(\Theta(n)\)

A, B – lists to be merged
L = new list; // empty
while (A not empty and B not empty )
\text{if } A.\text{first()} \leq B.\text{first()}
\then \text{ append}(L, A.\text{first()}); A = \text{rest}(A);
\text{else} \quad \text{append}(L, B.\text{first()}); B = \text{rest}(B);
\text{append}(L, A); // all remaining elements of A
\text{append}(L, B); // all remaining elements of B
return L
Run-time Analysis of Merge Sort

• Thus, the time required to sort an array of size \( n > 1 \) is:
  – the time required to sort the first half,
  – the time required to sort the second half, and
  – the time required to merge the two lists
• That is:

\[
T(n) = \begin{cases} 
\Theta(1) & n = 1 \\
2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1
\end{cases}
\]

Recursion tree

Solve \( T(n) = 2T(n/2) + cn \), where \( c > 0 \) is constant.

\[
\begin{array}{c}
\Theta(1) \\
\frac{\text{#leaves} = n}{\Theta(n)} \\
\end{array}
\]

Total = \( \Theta(n \log n) \)

Merge sort

• Worst case, average case, best case ...
  \( \Theta(n \log n) \)

• Common wisdom:
  – Requires additional space for merging (in case of arrays)
• Homework*: develop in-place merge of two lists implemented in arrays /compare speed/

Quicksort

• Divide-and-conquer algorithm.
• Sorts “in place” (like insertion sort, but not like merge sort).
• Very practical (with tuning).

Divide and conquer

Quick sort an \( n \)-element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** \( x \) such that elements in lower subarray \( \leq x \) \leq \text{elements in upper subarray.} \leq x \) \leq x \leq x

2. **Conquer:** Recursively sort the two subarrays.

3. **Combine:** Trivial.

Key: Linear-time partitioning subroutine.

Pseudocode for quicksort

\[
\begin{align*}
\text{QUICKSORT}(A, p, r) & \\
\text{if } p < r & \\
\text{then } q \leftarrow \text{PARTITION}(A, p, r) & \\
\text{QUICKSORT}(A, p, q-1) & \\
\text{QUICKSORT}(A, q+1, r) & \\
\end{align*}
\]

Initial call: \( \text{QUICKSORT}(A, 1, n) \)
### Partitioning subroutine

**PARTITION**($A, p, q$) $\rightarrow A[p..q]$

- $x$ $\rightarrow$ $A[p]$
- $i$ $\leftarrow$ $p$
- for $j$ from $p$ to $q$
  - if $A[j] \leq x$ then $i$ $\leftarrow$ $i + 1$
  - exchange $A[i] \leftrightarrow A[j]$
- return $i$

**Running time**

Running time $= O(n)$ for $n$ elements.

**Invariant:**

$\begin{array}{cccc}
  & p & i & j & q \\
\hline
\leq & x & \geq & x & ?
\end{array}$

September 21, 2005  Copyright © 2001–2 by Eyal D. Demaine and Charles E. Leiserson

---

### Partitioning version 2

- $i$ $=$ $L$; $j$ $=$ $R$ − 1;
- while $i < j$
  - while $(A[i] < pivot)$ $i$ $\leftarrow$ $i + 1$; // will stop at pivot latest
  - while $(i < j$ and $A[j]$ $\geq$ pivot) $j$ $\leftarrow$ $j$ − 1;
  - if $(i < j)$
    - $tmp$ $=$ $A[i]$; $A[i]$ $=$ $A[j]$; $A[j]$ $=$ $tmp$; $i$ $\leftarrow$ $i + 1$; $j$ $\leftarrow$ $j$ − 1
- $A[i]$ $=$ $pivot$;
- return $i$;

---

### Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$T(n) = T(0) + T(n - 1) + \Theta(n)$$

$$= \Theta(1) + T(n - 1) + \Theta(n)$$

$$= T(n - 1) + \Theta(n)$$

$$= \Theta(n^2)$$ (arithmetic series)

---

### Best-case analysis

**(For intuition only!)

If we’re lucky, **PARTITION** splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$

(same as merge sort)

What if the split is always $\frac{1}{10} \cdot \frac{9}{10}$?

$$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n)$$

What is the solution to this recurrence?

---

### Analysis of “almost-best” case

$$T(\frac{n}{10})$$

$$T(\frac{9n}{10})$$
### Analysis of “almost-best” case

- Choose a pivot:
  - Select the median of three ...
  - Select random – opponent cannot choose the winning strategy against you!

### More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ....

- \( L(n) = 2U(n/2) + \Theta(n) \) lucky
- \( U(n) = L(n-1) + \Theta(n) \) unlucky

Solving:

\[
L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\
= 2L(n/2 - 1) + \Theta(n) \\
= \Theta(n \log n) \text{ Lucky!}
\]

How can we make sure we are usually lucky?

### Randomized quicksort

**IDEA:** Partition around a random element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

### Randomized quicksort analysis

Let \( T(n) \) = the random variable for the running time of randomized quicksort on an input of size \( n \), assuming random numbers are independent.

For \( k = 0, 1, \ldots, n-1 \), define the indicator random variable

\[
X_i = \begin{cases} 
1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\
0 & \text{otherwise.}
\end{cases}
\]

\( E[X_i] = \Pr[X_i = 1] = 1/n \), since all splits are equally likely, assuming elements are distinct.
Analysis (continued)

\[ T(n) = \begin{cases} 
T(0) + T(n-1) + \Theta(n) & \text{if } 0 \leq n-1 \text{ split}, \\
T(1) + T(n-2) + \Theta(n) & \text{if } 1 \leq n-2 \text{ split}, \\
\vdots \\
T(n-1) + T(0) + \Theta(n) & \text{if } n-1 \leq 0 \text{ split}, \\
\end{cases} \]

\[ = \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \]

Calculating expectation

\[ E[T(n)] = \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \]

Linearity of expectation:

\[ E[X_k] = \frac{1}{n} \]

Calculating expectation

\[ E[T(n)] = \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \]

\[ = \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \]

\[ = \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k)] + \sum_{k=0}^{n-1} E[T(n-k-1)] + \sum_{k=0}^{n-1} E[\Theta(n)] \]

\[ = \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \]

Summations have identical terms.
We can sort in $O(n \log n)$

- Is that the best we can do?

- Remember: using comparisons $<$, $>$, $\leq$, $\geq$ we can not do better than $O(n \log n)$

How fast can we sort $n$ integers?

- E.g. sort people by year of birth?

- Sort people by sex?

Sorting in linear time

**Counting sort:** No comparisons between elements.

- **Input:** $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- **Output:** $B[1 \ldots n]$, sorted.
- **Auxiliary storage:** $C[1 \ldots k]$. 

Hairy recurrence

$E[T(n)] = \frac{1}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

**Prove:** $E[T(n)] \leq an \log n$ for constant $a > 0$.

- Choose $a$ large enough so that $an \log n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.

**Use fact:** $\sum_{k=2}^{n-1} k \log k = \frac{1}{2} n^2 \log n - \frac{1}{2} n^2$ (exercise).

Substitution method

$E[T(n)] \leq \frac{1}{n} \sum_{k=2}^{n-1} ak \log k + \Theta(n)$

$= \frac{2}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{2} n^2 \right) + \Theta(n)$

$= an \log n - \left( \frac{an}{4} - \Theta(n) \right)$

$\leq an \log n$,

if $a$ is chosen large enough so that $an/4$ dominates the $\Theta(n)$.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from code tuning.
- Quicksort behaves well even with caching and virtual memory.
### Counting sort

```plaintext
for i ← 1 to k  
do C[i] ← 0  
for j ← 1 to n  
do C[A[j]] ← C[A[j]] + 1  \(\Rightarrow C[i] = \{\text{key} = i\}\)  
for i ← 2 to k  
do C[i] ← C[i] + C[i-1]  \(\Rightarrow C[i] = \{\text{key} \leq i\}\)  
for j ← n downto 1  
do B[C[A[j]]] ← A[j]  
   \(\Rightarrow C[A[j]] ← C[A[j]] - 1\)
```

---

### Loop 1

<table>
<thead>
<tr>
<th>A:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

for i ← 1 to k  
do C[i] ← 0

---

### Loop 2

<table>
<thead>
<tr>
<th>A:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

for j ← 1 to n  
do C[A[j]] ← C[A[j]] + 1  \(\Rightarrow C[i] = \{\text{key} = i\}\)

---

### Loop 3

<table>
<thead>
<tr>
<th>A:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

for i ← 2 to k  
do C[i] ← C[i] + C[i-1]  \(\Rightarrow C[i] = \{\text{key} \leq i\}\)

---

### Loop 4

<table>
<thead>
<tr>
<th>A:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

for j ← n downto 1  
do B[C[A[j]]] ← A[j]  
   \(\Rightarrow C[A[j]] ← C[A[j]] - 1\)

---

### Analysis

\(\Theta(k)\)  
```
for i ← 1 to k  
do C[i] ← 0  
for j ← 1 to n  
do C[A[j]] ← C[A[j]] + 1  
for i ← 2 to k  
do C[i] ← C[i] + C[i-1]  
for j ← n downto 1  
do B[C[A[j]]] ← A[j]  
   \(\Rightarrow C[A[j]] ← C[A[j]] - 1\)
```

\(\Theta(n)\)

\(\Theta(n + k)\)
16.2.2012

**Radix sort**

- **Origin**: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix B.)
- Digit-by-digit sort.
- Hollerith’s original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on least-significant digit first with auxiliary stable sort.

**Operation of radix sort**

<table>
<thead>
<tr>
<th>Radix</th>
<th>329</th>
<th>720</th>
<th>720</th>
<th>329</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>457</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>839</td>
<td>839</td>
<td>839</td>
<td>839</td>
</tr>
<tr>
<td></td>
<td>436</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td></td>
<td>839</td>
<td>839</td>
<td>839</td>
<td>839</td>
</tr>
</tbody>
</table>

**Correctness of radix sort**

- **Induction on digit position**
  - Assume that the numbers are sorted by their low-order \( t-1 \) digits.
  - Sort on digit \( t \)
    - Two numbers that differ in digit \( t \) are correctly sorted.

**Running time**

If \( k = O(n) \), then counting sort takes \( \Theta(n) \) time.
- But, sorting takes \( \Omega(n \log n) \) time!
- Where’s the fallacy?

**Answer:**
- Comparison sorting takes \( \Omega(n \log n) \) time.
- Counting sort is not a comparison sort.
- In fact, not a single comparison between elements occurs!

**Stable sorting**

Counting sort is a **stable** sort: it preserves the input order among equal elements.

<table>
<thead>
<tr>
<th>A:</th>
<th>4</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Exercise:** What other sorts have this property?
Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order \( t - 1 \) digits.
- Sort on digit \( t \)
  - Two numbers that differ in digit \( t \) are correctly sorted.
  - Two numbers equal in digit \( t \) are put in the same order as the input ⇒ correct order.

Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort \( n \) computer words of \( b \) bits each.
- Each word can be viewed as having \( b/r \) base-2\(^r\) digits.

Example: 32-bit word

\[
\begin{array}{cccc}
8 & 8 & 8 & 8 \\
\end{array}
\]

\( r = 8 \Rightarrow b/r = 4 \) passes of counting sort on base-2\(^8\) digits; or \( r = 16 \Rightarrow b/r = 2 \) passes of counting sort on base-2\(^{16}\) digits.

How many passes should we make?

Analysis (continued)

Recall: Counting sort takes \( \Theta(n + k) \) time to sort \( n \) numbers in the range from 0 to \( k - 1 \).

If each \( b \)-bit word is broken into \( r \)-bit pieces, each pass of counting sort takes \( \Theta(n + 2^r) \) time.

Since there are \( b/r \) passes, we have

\[
T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right)
\]

Choose \( r \) to minimize \( T(n, b) \):

- Increasing \( r \) means fewer passes, but as \( r \gg \lg n \), the time grows exponentially.

Choosing \( r \)

\[
T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right)
\]

Minimize \( T(n, b) \) by differentiating and setting to 0.

Or, just observe that we don’t want \( 2^r \gg n \), and there’s no harm asymptotically in choosing \( r \) as large as possible subject to this constraint.

Choosing \( r = \lg n \) implies \( T(n, b) = \Theta(b n / \lg n) \).

- For numbers in the range from 0 to \( n^d - 1 \), we have \( b = d \lg n \Rightarrow \) radix sort runs in \( \Theta(d n) \) time.

Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):

- At most 3 passes when sorting \( \geq 2000 \) numbers.
- Merge sort and quicksort do at least \( \lceil \lg 2000 \rceil = 11 \) passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Radix sort using lists (stable)
16.2.2012

Radix sort using lists (stable)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbb</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>bbb</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>aac</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>ccc</td>
<td>green</td>
<td>green</td>
<td>green</td>
</tr>
</tbody>
</table>

Why not from left to right?

- Swap ‘0’ with first ‘1’
- Idea 1: recursively sort first and second half
  - Exercise ?

Bitwise sort left to right

- Idea2:
  - swap elements only if the prefixes match...
  - For all bits from most significant
    - advance when 0
    - when 1 > look for next 0
      - if prefix matches, swap
      - otherwise keep advancing on 0’s and look for next 1

Bitwise sort left to right

```c
void bitwisesort(SORTTYPE *ARRAY, int size) {
    int i, j, tmp, nibs;
    register SORTTYPE mask, curbit, group;
    nibs = sizeof(SORTTYPE) * 8;
    curbit = 1 << (nibs-1); /* set most significant bit 1 */
    mask = 0; /* mask of the already sorted area */
    do { /* For each bit */
        new_mask:
        for(i=0; i < size && !ARRAY[i] & curbit; i++) ;
        if(i >= size) goto array_end;
        group = ARRAY[i] & mask; /* Save current prefix snapshot */
        j = i;
        for( ; ; ) {
            if((i++ < size) && ARRAY[i] & mask) goto new_mask;
            if(!ARRAY[i] & curbit) {
                tmp = ARRAY[i];
                ARRAY[i] = ARRAY[j];
                ARRAY[j] = tmp;
                j += 1;
            }
        }
        mask = mask | curbit; /* area under mask is now sorted */
        curbit >>= 1; /* next bit */
    } while(curbit);
    /* all bits have been sorted... */
}
```

Jaak Vilo, Univ. of Tartu
Bitwise from left to right

0010000
0010010
0101000
0101100
1001010
1001001
1111000

• Swap ‘0’ with first ‘1’

Bucket sort

• Assume uniform distribution
• Allocate O(n) buckets
• Assign each value to pre-assigned bucket

Sort small buckets with insertion sort

http://sortbenchmark.org/

• Minutensort – max amount sorted in 1 minute
  – 116GB in 58.7 sec (Jim Wyllie, IBM Research)
  – 40-node 80-Itanium cluster, SAN array of 2,520 disks
• 2009, 500 GB Hadoop: 1406 nodes x (2 Quadcore Xeons, 8 GB memory, 4 SATA)
  Owen O’Malley and Arun Murthy
  Yahoo Inc.
• Performance / Price Sort and PennySort
Sort Benchmark

http://sortbenchmark.org/

We have a new benchmark called new GraySort, named in memory of the father of the sort benchmarks, Jim Gray. It replaces TeraByteSort which is now retired.

Unlike 2010, we will not be accepting early entries for the 2011 year. The deadline for submitting entries is April 1, 2011.

- All entries must be off-the-shelf and unmodified.
- For Daytona cluster sorts where input sampling is used to determine the output partition boundaries, the input sampling must be done evenly across all input partitions.

New rules for GraySort:

- The input file size is now minimum ~100TB or 1T records. Entries with larger input sizes also qualify.
- The winner will have the fastest SortedRecs/Min.
- We now provide a new input generator that works in parallel and generates binary data. See below.
- For the Daytona category, we have two new requirements: (1) The sort must run continuously (repeatedly) for a minimum 1 hour. (This is a minimum reliability requirement). (2) The system cannot overwrite the input file.

Order statistics

- Minimum – the smallest value
- Maximum – the largest value
- In general $i^{th}$ value.
- Find the median of the values in the array
- Median in sorted array $A$:
  - $n$ is odd: $A[(n+1)/2]$
  - $n$ is even: $A[(n+1)/2] \text{ or } A[(n+1)/2]$

Order stats:

- Input: A set $A$ of $n$ numbers and $i$, $1 \leq i \leq n$
- Output: $x$ from $A$ that is larger than exactly $i-1$ elements of $A$

Minimum

Minimum($A$)

1. $min = A[1]$
2. For $i = 2$ to length($A$)
3. If $min > A[i]$
4. Then $min = A[i]$
5. Return $min$

$n-1$ comparisons.

Min and max together

- compare every two elements $A[i], A[i+1]$
- Compare larger against current max
- Smaller against current min

$3 \lceil n / 2 \rceil$

Selection in expected $O(n)$

Randomised-select($A, p, r, i$)

If $p=r$ then return $A[p]$

$q = \text{Randomised-Partition}(A, p, r)$

$k = q - p + 1 \quad // \text{nr of elements in subarr}$

If $i = k$

then return $\text{Randomised-Partition}(A, q, p, i)$

else return $\text{Randomised-Partition}(A, q+1, r, i-k)$
Conclusion

• Sorting in general $O(\log n)$
• Quicksort is rather good
• Linear time sorting is achievable when one does not assume only direct comparisons
• Find $i^{th}$ value – expected $O(n)$
• Find $i^{th}$ value: worst case $O(n)$ – see CLRS
Skip List
A skip list, introduced by Pugh (1992), is a randomized balanced tree data structure organized as a series of increasingly sparse linked lists. Level 0 of a skip list is a linked list of all nodes in increasing order by key. For each level i greater than 0, each node in level i - 1 appears in level i independently with some fixed probability p. In a doubly linked skip list, each node stores a predecessor pointer and a successor pointer for each list in which it appears, for an average of $1/p$ pointers per node. The lists at the higher levels are the “shortcut levels” that allow the sequence of nodes to be traversed quickly. Searching for a node with a particular key involves searching first in the highest level, and repeatedly dropping down a level whenever it becomes clear that the node is not in the current level. Considering the search path in reverse shows that no more than $\frac{n}{p}$ nodes are searched on average per level, giving an average search time of $O(\log \frac{n}{p})$ with a node at level 0. Skip lists have been extensively studied (Pugh 1992; Papadimitriou et al. 1989; Demaine 1993; Kwek and Prentice 1990; Kwek and Prentice 1990); and because they require no global balancing operations are particularly useful in parallel systems (Graham et al. 1986; Graham and Marsaglia 1997).

Outline and Reading
• What is a skip list (§3.5)
• Operations
  — Search (§3.5.1)
  — Insertion (§3.5.2)
  — Deletion (§3.5.2)
• Implementation
• Analysis (§3.5.3)
  — Space usage
  — Search and update times

Skip lists
• Build several lists at different “skip” steps
• $O(n)$ list
• Level 1: $\sim \frac{n}{2}$
• Level 2: $\sim \frac{n}{4}$
• ... 
• Level $\log n \sim 2\ldots3$ elements...

Skip List
typedef struct nodeStructure *node;
typedef struct nodeStructure{
  keyType key;
  valueType value;
  node forward[1]; /* variable sized array of forward pointers */
};

Skip Lists

What is a Skip List
• A skip list for a set $S$ of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_k$ such that
  — Each list $S_i$ contains the keys of $S$ in nondecreasing order
  — Each list is a subsequence of the previous one, i.e.,
    $S_i \subseteq S_{i+1} \subseteq \ldots \subseteq S_k$
  — List $S_k$ contains only the two special keys
• We show how to use a skip list to implement the dictionary ADT
Search

- We search for a key \( x \) in a skip list as follows:
  - We start at the first position of the top list.
  - At the current position \( p \), we compare \( x \) with \( x \leftarrow \text{next}(p) \).
    - If \( x < x \leftarrow \text{next}(p) \), we go to the next position.
    - If \( x > x \leftarrow \text{next}(p) \), we “drop down”.
  - If we try to drop down past the bottom list, we return NO_SUCH_KEY.
- Example: search for 78.

**Algorithm:**

**Example:**

- **Insertion:**
  - To insert a key \( x \) into a skip list, we use a randomized algorithm:
    - We repeatedly toss a coin until we get tails, and we denote with \( i \) the number of times the coin came up heads.
    - If the coin is heads, we add the key to the top list \( S_0 \).
    - If the coin is tails, we add the key to the list \( S_i \) and search for the position \( p_0, p_1, \ldots, p_i \) of the key.
    - By Fact 1, the probability of getting \( \ell \) consecutive heads when flipping a coin is \( \frac{1}{2^{\ell}} \).

**Deletion:**

- To remove an item with key \( x \) from a skip list, we proceed as follows:
  - We search for a position \( p_0, p_1, \ldots, p_i \) of the item with key \( x \), where position \( p_j \) is in list \( S_j \).
  - We remove a pair of special keys from the lists \( S_0, S_1, \ldots, S_i \).
  - We remove all but one list containing only the two special keys.
- Example: remove key 34.

**Implementation:**

- We can implement a skip list with quad-nodes.
  - A quad-node stores:
    - item
    - link to the node before
    - link to the node after
    - link to the node above
  - Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them.

**Space Usage:**

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm.
  - We use the following two basic probabilistic facts:
    - **Fact 1:** The probability of getting \( \ell \) consecutive heads when flipping a coin is \( \frac{1}{2^{\ell}} \).
    - **Fact 2:** If each of \( n \) items is present in a set with probability \( p \), the expected size of the set is \( np \).
- Consider a skip list with \( n \) items.
  - By Fact 1, we insert an item into list \( S_0 \) with probability \( \frac{1}{2} \).
  - By Fact 2, the expected size of list \( S_0 \) is \( \frac{n}{2} \).
  - The expected number of nodes used by the skip list is

\[
\sum_{i=1}^{\infty} \frac{n}{2^i} = \frac{n}{2} \sum_{i=1}^{\infty} \frac{1}{2^i} < 2n
\]

Thus, the expected space usage of a skip list with \( n \) items is \( O(n) \).
15.2.2012

Height

- The running time of the search an insertion algorithm is affected by the height $h$ of the skip list.
- We show that with high probability, a skip list with $n$ items has height $O(\log n)$.
- We use the following additional probabilistic fact:

Fact 1: If each of $n$ events has probability $p$, the probability that at least one event occurs is at most $np$.

Fact 2: If a skip list with $n$ items has height $O(\log n)$, then the probability that list $\mathcal{L}$ has at least one item is at most $e^{-1}$.

Fact 3: By picking $p = \log n$, we have that the probability that $S_{\text{drop}}$ has at least one item is at most $e^{-1}$.

Fact 4: Thus a skip list with $n$ items has height at most $3\log n$ with probability at least $1 - \frac{1}{e}$.

Search and Update Times

- The search time in a skip list is proportional to
  - the number of drop-down steps, and
  - the number of scan-forward steps.

- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability.

- To analyze the scan-forward steps, we use yet another probabilistic fact.

Fact 4: The expected number of coin tosses required in order to get tails is 2.

- When we scan forward in a list, the destination key does not belong to a higher list.

Summary

- A skip list is a data structure for dictionaries that uses a randomized insertion algorithm.
- In a skip list with $n$ items
  - The expected space used is $O(n)$.
  - The expected search, insertion and deletion time is $O(\log n)$.

- Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability.

- Skip lists are fast and simple to implement in practice.

Conclusions

- Abstract data types hide implementations.
- Important is the functionality of the ADT.
- Data structures and algorithms determine the speed of the operations on data.
- Linear data structures provide good versatility.
- Sorting — a most typical need/algorithm.
- Sorting in $O(n \log n)$: Merge Sort, Quick sort.
- Solving recurrences — means to analyse.
- Skip lists — $\log n$ randomised data structure.