Algorithmics (6EAP)  
MTAT.03.238  
Linear structures, sorting, searching, Recurrences, Master Theorem...  
Jaak Vilo  
2019 Fall

Big-Oh notation classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Inferred</th>
<th>Intuition</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) \in \Theta(g(n))$</td>
<td>Bounded from above and below</td>
<td>“equal to”</td>
<td>$\approx$</td>
</tr>
<tr>
<td>$f(n) \in \Omega(g(n))$</td>
<td>Bounded from below</td>
<td>Lower bound</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$f(n) \in O(g(n))$</td>
<td>Bounded from above</td>
<td>Upper bound</td>
<td>$\leq$</td>
</tr>
<tr>
<td>$f(n) \in o(g(n))$</td>
<td>F is dominated by g</td>
<td>Strictly below</td>
<td>$&lt;$</td>
</tr>
</tbody>
</table>

Linear, sequential, ordered, list ...

Physical ordered list ~ array

- Memory /address/  
  - Garbage collection
- Files (character/byte list/lines in text file,...)
- Disk  
  - Disk fragmentation

Linear data structures: Arrays

- Array
- Bidirectional map
- Bit array
- Bit field
- Bitboard
- Bitmap
- Circular buffer
- Control table
- Image
- Dynamic array
- Gap buffer
- Hashed array tree
- Heightmap
- Lookup table
- Matrix
- Parallel array
- Sorted array
- Sparse array
- Sparse matrix
- Iliffe vector
- Variable-length array

Linear data structures: Lists

- Doubly linked list
- Array list
- Linked list
- Self-organizing list
- Skip list
- Unrolled linked list
- VList
- Xor linked list
- Zipper
- Doubly connected edge list
- Difference list
Lists: Array

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

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

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

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

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** (a 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

Abstract Data Types (ADT)

- Some common ADTs, which have proved useful in a great variety of applications, are:

  **Container**
  - **List**
  - **Set**
  - **Multiset**
  - **Map**
  - **Multimap**

  **Stack**
  - **Graph**
  - **Queue**
  - **Priority queue**
  - **Double-ended queue**
  - **Double-ended priority queue**

- Each of these ADTs may be defined in many ways and variants, not necessarily equivalent.

Abstract data types:

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

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

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**
### Linear data structures

**Arrays**
- Array
- Bidirectional map
- Bit array
- Bit field
- Bitboard
- Bitmap
- Circular buffer
- Control table
- Image
- Dynamic array
- Gap buffer

**Lists**
- Doubly linked list
- Linked list
- Self-organizing list
- Skip list
- Unrolled linked list
- VList
- Xor linked list
- Zipper
- Doubly connected edge list

### Trees ...

**Binary trees**
- AVL tree
- Balanced tree
- Binary search tree
- Binary heap
- Binomial heap
- Fibonacci heap

**Heaps**
- Fibonacci heap
- Binomial heap
- Binary heap

**Hashes, Graphs, Other**

**Hashes**
- Bloom filter
- Distributed hash table
- Hash array
- Hash table
- Hash tree
- Hash trie
- Koorde
- 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

**Arrays**

**Lists: Array**

- Access
- Insert to end
- Delete from end
- Insert
- Delete
- Search

```
<table>
<thead>
<tr>
<th>size</th>
<th>0 1</th>
<th>3 6 7 8 5 2</th>
</tr>
</thead>
</table>
```

- Insert 8 after L[2]
- Delete last

**Lists: Array**

```
<table>
<thead>
<tr>
<th>size</th>
<th>0 1</th>
<th>3 6 7 8 5 2</th>
</tr>
</thead>
</table>
```

- 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
Linked lists

- Singly linked
- Doubly linked

Linked lists: add

Linked lists: delete (+ garbage collection?)

Introduction to linked lists

- Consider the following struct definition
  ```
  struct node {
    string word;
    int num;
    node *next; //pointer for the next node
  };
  node *p = new node;
  ```

Introduction to linked lists: inserting a node

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

- p -> 5 -> Ali -> NULL

Array indexed from 0 to n – 1:

<table>
<thead>
<tr>
<th>Access/Change the kth element</th>
<th>Insert before or after the kth element</th>
<th>Delete the kth element</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Singly-linked list with head and tail pointers

<table>
<thead>
<tr>
<th>Access/Change the kth element</th>
<th>Insert before or after the kth element</th>
<th>Delete the kth element</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(k)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Doubly linked list

<table>
<thead>
<tr>
<th>Access/Change the kth element</th>
<th>Insert before or after the kth element</th>
<th>Delete the kth element</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(k)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Introduction to linked lists: adding a new node

- How can you add another node that is pointed by \( p \rightarrow \text{link} \)?
- \( \text{node} *p; \)
- \( p = \text{new node}; \)
- \( p->\text{num} = 5; \)
- \( p->\text{word} = "\text{Ali}"; \)
- \( p->\text{next} = \text{NULL}; \)
- \( \text{node} *q; \)

```
5  Ali
num word link
```

```
5  Ali
num word link
p     q
```

Pointers in C/C++

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

Book-keeping

- \text{malloc, new} – “remember” what has been created \text{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 \text{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


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

```
head=3
next key prev
/ / / / / / / /
1 2 3 4 5 6 7
/ / / / / / / /
1 2 3 4 5 6 7
```

```
head
/ / / / / / / /
1 2 3 4 5 6 7
```

```
free = -1 => array is full
allocate object:
new = free;
free = next[free];
free object x:
next[x]= free
free = x
```

Maintaining list of free objects

```
head=3
next key prev
/ / / / / / / /
1 2 3 4 5 6 7
/ / / / / / / /
1 2 3 4 5 6 7
```

```
head=3 free=6
next key prev
/ / / / / / / /
1 2 3 4 5 6 7
/ / / / / / / /
1 2 3 4 5 6 7
```

Multiple lists, single free list

```
head=3
next key prev
/ / / / / / / /
1 2 3 4 5 6 7
/ / / / / / / /
1 2 3 4 5 6 7
```

```
head=6
next key prev
/ / / / / / / /
1 2 3 4 5 6 7
/ / / / / / / /
1 2 3 4 5 6 7
```

```
free = 2
```

```
free = -1 => array is full
allocate object:
new = free;
free = next[free];
free object x:
next[x]= free
free = x
```

```
free = 6
```

```
head=7
next key prev
/ / / / / / / /
1 2 3 4 5 6 7
/ / / / / / / /
1 2 3 4 5 6 7
```

```
head=9
next key prev
/ / / / / / / /
1 2 3 4 5 6 7
/ / / / / / / /
1 2 3 4 5 6 7
```

```
free = 3, 9
```

```
free = 3, 9
```

```
(2)
```
Hack: allocate more arrays ...

use integer division and mod

AA[ (i-1)7 ] := ( (i-1) % 7 )

LIST(10) = AA[ 1 ][ 2 ]
LIST(19) = AA[ 2 ][ 5 ]

Queue

- enqueue(x) - add to end
- dequeue() - fetch from beginning

- FIFO – First In First Out

- O(1) in all reasonable cases 😊

Queue (FIFO)

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

```
First = List[F]
Last = List[(L+1)%MAX_SIZE]
```

**Add to end** $(x)$:
```
{ List[L]=x; L=(L+1)%MAX_SIZE }
```

**Full**: $\text{return } ((L+1)%\text{MAX SIZE} == F)$

**Empty**: $F==L$

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 ever be?**

```
1 6 7 5
1 6 7 5 2
```

- When full, allocate bigger table dynamically, and copy all previous values there

- $O(n)$?

What about deletions?

- When full, create 2x bigger table, copy previous n elements:

- After every $2^k$ insertions, perform $O(n)$ copy

- $O(n)$ individual insertions +

- $n/2 + n/4 + n/8 ...$ copy-ing

- Total: $O(n)$ effort!
What about deletions?

- 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) => insert(k,D) = $O(1)$ or $O(n)$ – test for uniqueness

Array based sorted list

- is d in D ?
- Binary search in D

---

Binary search – recursive

```c
int BinarySearch(int A[0..N-1], int value, int low, int high) {
    if (high < low)        // not found
        return -1
    mid = (low + high) / 2
    if (A[mid] > value)    // Note: not (low + high) / 2 !!
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid  // found
}
```

Binary search – recursive

```c
int BinarySearch(int A[0..N-1], int value, int low, int high) {
    if (high < low)        // not found
        return -1
    mid = low + (high - low) / 2  // Note: not (low + high) / 2 !!
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid  // found
}
```
Binary search – *iterative*

```cpp
BinarySearch(A[0..N-1], value) {
    low = 0; high = N - 1;
    while (low <= high) {
        mid = low + ((high - low) / 2) // Note: not (low + high) / 2 !!
        if (A[mid] > value) 
            high = mid - 1
        else if (A[mid] < value) 
            low = mid + 1
        else 
            return mid // found
    }
    return -1 // not found
}
```

**Work performed**
- $x \leftrightarrow A[18] \ ? \ <$
- $x \leftrightarrow A[9] \ ? \ >$
- $x \leftrightarrow A[13] \ ? \ ==$
- $O(\lg n)$

**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...

**Decision-tree example**

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

```
2 3
\_\_\_
2 1\_\_\_
1 2 3
1 3
1 2 3
2 3
1 2 3
```

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.

**Lower bound for decision-tree sorting**

**Theorem.** Any decision tree that can sort $n$ elements must have height $\Omega(n \log n)$.

**Proof.** The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-$h$ binary tree has $\leq 2^n$ leaves. Thus, $n! \leq 2^n$.

$$h \geq \lg(n!) \geq \lg((n/e)^n) = n \lg n - n \lg e = \Omega(n \lg n).$$
Proof: \( \log(n!) = \Omega(n \log n) \)

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

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

\[
\lceil \log_2 n \rceil \geq \log_2 n!
\]
\[ \geq \sum_{i=1}^{n/2} \log_2 i \]
\[ \geq \sum_{i=1}^{n/2} \log_2 n/2 \]
\[ \geq n/2 \log_2 n/2 \]
\[ = \Omega(n \log n). \]

Example

- Applying the merge sort algorithm:

```plaintext
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 )
  if A.first() \( \leq \) B.first()
    then  append( L, A.first() ) ; A = rest(A) ;
    else  append( L, B.first() ) ; B = rest(B) ;
  append( L, A);  // all remaining elements of A
  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:
  $$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + o(n) & n > 1 \end{cases}$$

Recursion tree

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

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/

Quick sort

- 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 \(\leq x\) are elements in upper subarray \(\geq x\).
2. **Conquer**: Recursively sort the two subarrays.
3. **Combine**: Trivial.

Key: Linear-time partitioning subroutine.

Pseudocode for quicksort

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

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

Partitioning subroutine

\[
\text{PARTITION}(A, p, q) \\
x \leftarrow A[p] \\
p \leftarrow p \\
\text{for } j \leftarrow p+1 \text{ to q} \\
\text{do if } A[j] \leq x \\
\text{then } i \leftarrow i + 1 \\
\text{exchange } A[i] \leftrightarrow A[j] \\
\text{return } i
\]

Invariant: \(x \leq x \leq x\)

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

Partitioning version 2

\[
\text{pivot } = A[R]; \\
i=\text{L}; j=\text{R}-1; \\
\text{while } (i \leq j) \\
\text{do if } A[i] < \text{pivot} \text{ then } i++; \\
\text{else} \\
\text{do if } A[j] > \text{pivot} \text{ then } j--; \\
\text{else} \\
\text{exchange } A[i] \leftrightarrow A[j]; \\
i++; j--; \\
\text{return } i
\]

Worst-case of quicksort

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

\[
T(n) = T(0) + T(n-1) + \Theta(n) \\
= \Theta(1) + T(n-1) + \Theta(n) \\
= T(n-1) + \Theta(n) \\
= \Theta(n^2) \text{ (arithmetic series)}
\]
Choice of pivot in Quicksort

- Select median of three ...
- Select random – opponent can not choose the winning strategy against you!

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.

Analysis of “almost-best” case

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

\[ T(n) = 2T(n/2) + \Theta(n) = O(n \log n) \]  
(same as merge sort)

What if the split is always \( \frac{1}{10} \) or \( \frac{9}{10} \) ?

\[ T(n) = T(\frac{1}{10} n) + T(\frac{9}{10} n) + \Theta(n) \]

What is the solution to this recurrence?

Analysis of “almost-best” case

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:

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

How can we make sure we are usually lucky?
Random pivot

Select pivot randomly from the region (blue) and swap with last position
Select pivot as a median of 3 [or more] random values from region

Apply non-recursive sort for array less than 10-20

- 2-pivot version of Quicksort — split in 3 regions!
- Handle equal values (equal to pivots) as „third region“

Bentley-McIlroy 3-way partitioning

<table>
<thead>
<tr>
<th>Partitioning invariant</th>
<th>equal</th>
<th>less</th>
<th>greater</th>
<th>equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>move from left to find an element that is not less</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>move from right to find an element that is not greater</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stop if pointers have crossed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exchange</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if left element equal, exchange to left end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if right element equal, exchange to right end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap equals to center after partition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

KEY FEATURES
- always uses N-1 (three-way) compares
- no extra overhead if no equal keys
- only one “extra” exchange per equal key

Robert Sedgewick
Jan Bentley

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.

Randomized quicksort analysis

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

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

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

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

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

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

Calculating expectation

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

Take expectations of both sides.

Calculating expectation

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

Linearity of expectation.

Calculating expectation

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

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

Independence of \( X_k \) from other random choices.

Calculating expectation

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

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

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

Summations have identical terms.

Linearity of expectation; \( E[X] = 1/n \).
Hairy recurrence

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

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

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

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

Use fact: \( \sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \) (exercise).

Substitution method

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

\[ = 2a \left( \frac{1}{n} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \]

\[ = an \lg n - \left( \frac{an}{4} - \Theta(n) \right) \]

\[ \leq an \lg n , \] if \( a \) is chosen large enough so that \( an/4 \) dominates the \( \Theta(n) \).

Alternative materials

- Quicksort average case analysis
  http://eid.ee/10z
  - [MIT Open Courseware - Asymptotic notation, Recurrences, Substituton Master Method](http://eid.ee/10y)
- http://eid.ee/10z

The master method

The master method applies to recurrences of the form

\[ T(n) = a T(n/b) + f(n) \]

where \( a \geq 1 \), \( b > 1 \), and \( f \) is asymptotically positive.
Three common cases

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an $n^{-\epsilon}$ factor).
   
   **Solution:** $T(n) = \Theta(n^{\log_b a})$.

Three common cases (cont.)

Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$.
   - $f(n)$ grows exponentially faster than $n^{\log_b a}$ (by an $n^{+\epsilon}$ factor),
   - and $f(n)$ satisfies the regularity condition that $af(n/b) \leq cf(n)$ for some constant $c < 1$.
   
   **Solution:** $T(n) = \Theta(f(n))$.

Examples

**EX.** $T(n) = 4T(n/2) + n^3$
- $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$.
   - **Case 3:** $f(n) = \Omega(n^{2 + \epsilon})$ for $\epsilon > 0$
   - and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.
   - $\therefore T(n) = \Theta(n^3)$.

**Examples**

**EX.** $T(n) = 4T(n/2) + n^3$
- $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$.
   - **Case 3:** $f(n) = \Omega(n^{2 + \epsilon})$ for $\epsilon > 0$
   - and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.
   - $\therefore T(n) = \Theta(n^3)$.

**EX.** $T(n) = 4T(n/2) + n^2 \log n$
- $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2 \log n$.
   - Master method does not apply. In particular, for every constant $\epsilon > 0$, we have $n^\epsilon = o(n \log n)$.
**Idea of master theorem**

**Recursion tree:**

- $f(n)$
- $f(n/b)$
- $f(n/b^2)$
- $f(n/b^3)$
- $\ldots$
- $f(n/b^h)$
- $f(n)$
- $a f(n/b)$
- $a^2 f(n/b^2)$
- $\ldots$
- $a^h f(n/b^h)$
- $T(1)$

- $h = \log_b n$
- #leaves $= a^h$
- $= \Omega(\log_{b^2} n)$
- $= \Omega(\log_b n)$

**CASE 1:** The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

$\Theta(n \log_{b^2} n) T(1)$

**CASE 2:** ($k = 0$) The weight is approximately the same on each of the $\log_{b^2} n$ levels.

$\Theta(n \log_{b^2} n) T(1)$

**CASE 3:** The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

$\Theta(f(n))$
Back to sorting

We can sort in $O(n \log n)$

- using comparisons ($<$ $<=$ $>$ $>$) we cannot do better than $O(n \log n)$
- Is that the best we can do?

How fast can we sort $n$ integers?

- Sort people by their sex? (F/M, 0/1)
- Sort people by year of birth?

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:** $C[1 \ldots k]$.

### 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 ▷ C[i] = |\{key = i\}|
for i ← 2 to k
    do C[i] ← C[i] + C[i-1] ▷ C[i] = |\{key ≤ i\}|
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
        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>C:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>B:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
**Loop 2**

\[
A: \begin{bmatrix} 4 & 1 & 3 & 4 & 3 \end{bmatrix} \quad C: \begin{bmatrix} 1 & 0 & 2 & 2 \end{bmatrix}
\]

\[
B: \begin{bmatrix} \phantom{1} & \phantom{1} & \phantom{1} & \phantom{1} & \phantom{1} \end{bmatrix} \quad C': \begin{bmatrix} 1 & 1 & 3 & 5 \end{bmatrix}
\]

for \( j \leftarrow 1 \) to \( n \)

\[
do C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{key = i\}|
\]

**Loop 3**

\[
A: \begin{bmatrix} 4 & 1 & 3 & 4 & 3 \end{bmatrix} \quad C: \begin{bmatrix} 1 & 0 & 2 & 2 \end{bmatrix}
\]

\[
B: \begin{bmatrix} \phantom{1} & \phantom{1} & \phantom{1} & \phantom{1} & \phantom{1} \end{bmatrix} \quad C': \begin{bmatrix} 1 & 1 & 3 & 5 \end{bmatrix}
\]

for \( i \leftarrow 2 \) to \( k \)

\[
do C[i] \leftarrow C[i] + C[i-1] \quad \triangleright \quad C[i] = |\{key \leq i\}|
\]

**Loop 4**

\[
A: \begin{bmatrix} 4 & 1 & 3 & 4 & 3 \end{bmatrix} \quad C: \begin{bmatrix} 1 & 1 & 2 & 5 \end{bmatrix}
\]

\[
B: \begin{bmatrix} 3 & \phantom{1} & \phantom{1} & \phantom{1} \end{bmatrix} \quad C': \begin{bmatrix} 1 & 1 & 2 & 4 \end{bmatrix}
\]

for \( j \leftarrow n \) downto \( 1 \)

\[
\]

**Analysis**

\[
\Theta(k) \quad \begin{cases} 
\text{for } i \leftarrow 1 \text{ to } k \\
\quad \text{do } C[i] \leftarrow 0 \\
\text{for } j \leftarrow 1 \text{ to } n \\
\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
\text{for } i \leftarrow 2 \text{ to } k \\
\quad \text{do } C[i] \leftarrow C[i] + C[i-1] \\
\text{for } j \leftarrow n \text{ downto } 1 \\
\quad \text{do } B[C[A[j]]] \leftarrow A[j] \\
\quad \text{C[A[j]]} \leftarrow C[A[j]] - 1 \\
\end{cases}
\]

\[
\Theta(n + k)
\]

**Running time**

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

**Answer:**
- *Comparison sorting* takes \( \Omega(n \lg n) \) 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.

\[
A: \begin{bmatrix} 4 & 1 & 3 & 4 & 3 \end{bmatrix} \\
B: \begin{bmatrix} 1 & 3 & 3 & 4 & 4 \end{bmatrix}
\]

**Exercise:** What other sorts have this property?
Radix sort

- **Origin**: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix 6.)
- 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.

Radix sort

Radix-Sort(A,d)

1. for i = 1 to d /* least significant to most significant */
2. use a stable sort to sort A on digit i

Operation of radix sort

<table>
<thead>
<tr>
<th>329</th>
<th>720</th>
<th>720</th>
<th>329</th>
</tr>
</thead>
<tbody>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
<td>355</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
<td>457</td>
</tr>
<tr>
<td>436</td>
<td>657</td>
<td>355</td>
<td>657</td>
</tr>
<tr>
<td>720</td>
<td>329</td>
<td>457</td>
<td>720</td>
</tr>
<tr>
<td>355</td>
<td>839</td>
<td>657</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.
  - 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-2r digits.

**Example**: 32-bit word:

```
| 88 | 88 | 88 |
```

\( r = 8 \Rightarrow b/r = 4 \) passes of counting sort on base-28 digits; or \( r = 16 \Rightarrow b/r = 2 \) passes of counting sort on base-216 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 $r$ can be done by differentiation 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 $T(n, b) = \Theta(bn/\lg n)$.

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

• Idea 2:
  – 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 from left to right

• Swap ‘0’ with first ‘1’

Bucket sort

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

Bitwise left to right sort

/* Historical sorting - was used in Univ. of Tartu using assembler... */
/* C implementation - Jaak Vilo, 1989 */

void bitwisesort(SORTTYPE *ARRAY, int size)
{
  int i, j, tmp, nrbits;
  register SORTTYPE mask, curbit, group;
  nrbits = sizeof(SORTTYPE) * 8;
  curbit = 1 << (nrbits - 1); /* set most significant bit 1 */
  mask = 0;
  do {                      /* For each bit */
    i = 0;
    new_mask:
    for( ; (i < size) && (ARRAY[i] & curbit) == 0; i++);
    /* Advance while bit == 0 */
    if( i == size )
      goto array_end;
    group = ARRAY[i] & mask;
    /* Save current prefix snapshot */
    j = i;
    /* Memorize location of 1 */
    for( ; ; )
    {
      i++;
      /* Reached end of array */
      if( (ARRAY[i] & mask) != group)
        goto new_mask;      /* new prefix */
      if( (ARRAY[i] & curbit) == 0 )
      {    /* bit is 0 - need to swap with previous location of 1, ARRAY[j] */
        tmp = ARRAY[i];
        ARRAY[i] = ARRAY[j];
        ARRAY[j] = tmp;
        j++;
      } /* Swap and increase j to the next possible 1 */
    }
  } while( curbit );
  array_end:
  mask = mask | curbit;
  /* area under mask is now sorted */
  curbit >>= 1;          /* next bit */
}  while( curbit );
/* until all bits have been sorted... */

Jaak Vilo, Univ. of Tartu
Sort small buckets with insertion sort

http://sortbenchmark.org/

- The sort input records must be 100 bytes in length, with the first 10 bytes being a random key
- Minutesort – 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/](http://sortbenchmark.org/)
- Sort Benchmark Home Page
- We have a new benchmark called GraySort, 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 hardware used 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 uniformly 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 (uninterrupted) 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 \( A[(n+1)/2] \) or \( A[(n+1)/2] \)

Q: Find i'th value in unsorted data

A. \( O(n) \)
B. \( O(n \log \log n) \)
C. \( O(n \log n) \)
D. \( O(n \log^2 n) \)

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

\[ 3 \lceil \frac{n}{2} \rceil \]

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 = $q - p + 1$ // nr of elements in subarr
if $i \leq k$
then return Randomised-Partition($A, p, q, i$)
else return Randomised-Partition($A, q+1, r, i-k$)

Conclusion

- Sorting in general $O(n \log n)$
- Quicksort is rather good
- Linear time sorting is achievable when one does not assume only direct comparisons
- Find $i$’th value in unsorted – expected $O(n)$
- Find $i$’th value: worst case $O(n)$ – see CLRS

Ok...

- lists – a versatile data structure for various purposes
- Sorting – a typical algorithm (many ways)
- Which sorting methods for array/list?
- Array: most of the important (e.g. update) tasks seem to be $O(n)$, which is bad

Q: search for a value $X$ in linked list?

A. $O(1)$
B. $O(\log n)$
C. $O(n)$

Can we search faster in linked lists?

- Why sort linked lists if search anyway $O(n)$?
- Linked lists:
  - what is the “mid-point” of any sublist?
  - Therefore, binary search can not be used...
- Or can it?
Skip List

A skip list, introduced by Pagh (Pagh, 1999), is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists. Level 0 of a skip list is a simple list of all nodes in increasing order by key. For each i greater than 0, each node in level i-1 appears in level i independently with some fixed probability p. In a doubly-lifted skip list, each node stores a predecessor pointer and a successor pointer for each list in which it appears, for an average of $\frac{1}{2}$ pointers per node. The lists at the higher level act as “express lanes” that allow the search of nodes to be traversed quickly. Searching for a node with a specific key k is performed by starting at head and proceeding through its lists until reaching a list whose keys are greater than or equal to k. If the node is not in the current level, the search is continued in the list of the next lower level. This process is repeated until the node is found or the search reaches the bottom list.

Figure 1. A skip list with n = 8 nodes and log n = 3 levels.

Skip Lists

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

What is a Skip List

- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that
  - Each list $S_i$ contains the special keys +\infty and -\infty.
  - List $S_0$ contains the keys of $S$ in nondecreasing order.
  - Each list is a subsequence of the previous one, i.e., $S_0 \subseteq S_1 \subseteq \cdots \subseteq S_h$.
  - List $S_h$ contains only the two special keys.
- We show how to use a skip list to implement the dictionary ADT.

Illustration of lists

Key
- Right[ .. ] - right links in array
- Left[ .. ] - left links in array, array size - how high is the list

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 2-3 elements...
Search
- We search for a key $x$ in a skip list as follows:
  - Start at the first position of the top list
  - At the current position $p$, we compare $x$ with $y = \text{key}(p)$
  - If $x > y$, we “scan forward”
  - If $x < y$, we “drop down”
- Example: search for 78

Insertion
- To insert an item $(x, v)$ into a skip list, we use a randomized algorithm:
  - In each list $S_0, \ldots, S_i$, each containing only the two special keys,
  - Search for $x$ in the list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than $x$ in each list $S_0, S_1, \ldots, S_i$
  - For $j = 0, \ldots, i$, insert $x, v$ into list $S_j$ after position $p_j$
- Example: insert key 15, with $i = 2$

Randomized Algorithms
- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type $b = \text{random}()$
- Its running time depends on the outcomes of the coin tosses

Deletion
- To remove an item with key $x$ from a skip list, we proceed as follows:
  - Search for $x$ in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key $x$, where position $p_j$ is in list $S_j$
  - Remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - Remove all but one list containing only the two special keys.
- Example: remove key 34

Implementation v2
- We can implement a skip list with quad-nodes
- A quad-node stores:
  - An item
  - Link to the node before
  - Link to the node after
  - Link to the node below
  - 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 $i$ consecutive heads when flipping a coin is $2^{-i}$
  - 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 in list $S_j$ with probability $1/2^j$
  - By Fact 2, the expected size of list $S_j$ is $2^j$
- The expected number of nodes used by the skip list is $\sum_{j=0}^{\infty} \frac{1}{2^j} \cdot \frac{1}{2^j} < 2n$
- Thus, the expected space usage of a skip list with $n$ items is $O(n)$
**Height**

- The running time of the search an insertion algorithms 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 3:** If each of $n$ events has probability $p$, the probability that at least one event occurs is at most $np$.

- Consider a skip list with $n$ items:
  
  - By Fact 1, we insert an item in list $S_i$ with probability $\frac{1}{2^{i-1}}$.
  
  - By Fact 3, the probability that list $S_i$ has at least one item is at most $\frac{1}{2^{i-1}}$.
  
  - Thus, a skip list with $n$ items has height at most $\frac{n}{2}$.

- By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one item is at most $\frac{n}{2^{3\log n}} = \frac{n}{2^{\log n^2}} = \frac{1}{n^{1/2}}$.

- Thus, a skip list with $n$ items has height at most $1 - \frac{1}{n^{1/2}}$.

**Search and Update Times**

- The search time in a skip list is proportional to:
  
  - the number of drop-down steps,
  
  - plus 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 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.

- A scan-forward step is associated with a former coin toss that gave tails.

- By Fact 4, in each list the expected number of scan-forward steps is $2$.

- Thus, the expected number of scan-forward steps is $O(\log n)$.

- We conclude that a search in a skip list takes $O(\log n)$ expected time.

- The analysis of insertion and deletion gives similar results.

**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, Quicksort.

- Solving Recurrences – means to analyse.

- Skip lists – $\log n$ randomised data structure.