Linear, sequential, ordered, list ...

Memory, disk, tape etc – is an ordered sequentially addressed media.

Physical ordered list ~ array

- Memory /address/
  - Garbage collection

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

- Disk
  - Disk fragmentation

Lists: Array

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

Lists: Array

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

2D array

\[ A[i][j] = A + i \times (nr \_ el \_ in \_ row \times el \_ size) + j \times el \_ size \]
21.2.2013

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

1 2 3

4 5 6

7 8 9

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 iff 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

Abstract data types:

• Dictionary (key,value)
• Stack (LIFO)
• Queue (FIFO)
• Queue (double-ended)
• Priority queue (fetch highest-value 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
**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/Lists
- Ordered
  - Array
  - Sequence (Skip Lists)
  - Binary Search Tree [BST]
  - AVL trees, red-black trees
  - (2, 4) Trees
  - B-Trees
- Valued
  - Hash Tables
  - Extensible 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 (a larger size of type float)
- String (for string of chars)
- 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 buffer
- Control table
- Image
- Dynamic array

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

**Trees**

- Binary tree
- AVL tree
- Red-Black tree
- B-tree
- B+ tree
- hi B tree
- B*-tree
- BXM tree
- R tree
- R* tree
- UB-tree
- Z-order
- Linear
- Octree
- KD tree
- KD tree
- KD tree

**Hashes, Graphs, Other**

**Hashes**
- Bloom filter
- Distributed hash table
- Hash array mapped trie
- Hash list
- Hash table
- Hash tree
- Koaide
- 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

**Other**
- Lightmap
- Winged edge
- Quad edge
- Routing table
- Symbol table
Lists: Array

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

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

- \texttt{push(x)} -- add to end (add to top)
- \texttt{pop()} -- fetch from end (top)

- \(O(1)\) in all reasonable cases

- LIFO -- Last In, First Out

Linked lists

- Singly linked
- Doubly linked
Linked lists: delete (+ garbage collection?)

Doubly-linked list:

Singly-linked list:

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

Operations

• Array indexed from 0 to $n - 1$:

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

*under the assumption we have a pointer to the $k$th node. $O(n)$ otherwise

• Doubly-linked list

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

Improving Run-Time Efficiency

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

Singly-linked list:

Doubly-linked list:

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

Improving Efficiency

Singly-linked list:

Doubly-linked list:

- under the assumption we have a pointer to the $k$th node, $O(n)$ otherwise

Array indexed from 0 to $n - 1$:

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

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;
```

21.2.2013
Introduction to linked lists: inserting a node

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

```
5  Ali

num   word   next
```

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;
- q = new node;
- q->num = 8;
- q->word = "Veli";
- p->next = q;
- q->next = NULL;

```
5  Ali

num   word   link

7
```

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";
p->next = q;
q->next = NULL;
```

```
5  Ali

num   word   link

7
```

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/
- C++ Memory Management : What is the difference between malloc/free and new/delete?

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

- head=3
- free=6
- free = -1 => array is full
- allocate object:
  - new = free;
  - free = next[free];
- free object x:
  - next[x]= free
  - free = x
**Multiple lists, single free list**

- head1=3 => 8, 4, 7
- head2=6 => 3, 9
- free = 2

<table>
<thead>
<tr>
<th>next</th>
<th>key</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

**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]

**XOR linked lists**

- A, B, C, D, E
- next
- prev

- An XOR linked list compresses the same information into one address field by storing the bit-wise XOR of the address for previous and the address for next in one field.

- ... ABC <--> BCD <--> CDE <--> DAE ...

- When you traverse the list from left to right: supposing you are at C, you can take the address of the previous item, B, and XOR it with the value in the link field (B|D). You will then have the address for D and you can continue traversing the list. The same pattern applies in the other direction.

**Queue (FIFO)**

- F
- L

- First = List[F]
- Last = List[L-1]
- Full: return (L = MAX_SIZE )
- Empty: F < 0 or F >= L

**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]
Add_to_end(x) : { List[L]=x ; L= (L+1) % MAX_SIZE } // % = modulo
Last = List[(L+1) % MAX_SIZE] % MAX_SIZE ]
Full: return ((L+1)%MAX_SIZE == F)
Empty: F==L

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 + => 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) ?

• 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!
21.2.2013

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

- 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

Array based sorted list

- is d in D?
- Binary search in D

Binary search / recursive

```c
BinarySearch(A[0..N-1], value, low, high)
{
    if (high < low)
        return -1 // not found
    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

```c
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

- $x \leftrightarrow A[18] \ ? \ <$
- $x \leftrightarrow A[9] \ ? \ >$
- $x \leftrightarrow A[13] \ ? \ ==$

- $O(\log n)$

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...

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

- $\log(n!) = \log n + \log(n-1) + \log(n-2) + \ldots + \log(1)$
- $\geq n/2 \times \log(n/2)$
- $= \Omega(n \log n)$

Decision tree model

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

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.

$[\log_2 n!] \geq \log_2 n!
\geq \sum_{i=1}^{n/2} \log_2 i
\geq \sum_{i=1}^{n/2} \log_2 n/2
\geq n/2 \log_2 n/2
= \Omega(n \log n).$
• \( \log(n!) = \log(n)+\log(n-1)+\ldots+\log(1) \)
  
  a) \( \leq n \log(n) \)
  
  b) \( \geq n/2 \times \log(n/2) = n/2 \log n - n/2 \)

**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**

\[ \text{Merge-Sort}(A, p, r) \]

\[
\begin{align*}
\text{if } & p < r \\
\text{then } & q = (p+r)/2 \\
\text{Merge-Sort}(A, p, q) \\
\text{Merge-Sort}(A, q+1, r) \\
\text{Merge}(A, p, q, r) \\
\end{align*}
\]

It was invented by *John von Neumann* in 1945.

**Example**

- Applying the merge sort algorithm:

  ![Diagram of merge sort example]

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

\( A, B \) – lists to be merged

\( L = \text{new list}; // empty \)

\( \text{while( } A \text{ not empty and } B \text{ not empty )} \)

\( \text{if } A\text{.first()} \leq B\text{.first()} \)

\( \text{then append( } L, A\text{.first()} \) ; \( A = \text{rest}(A) \) ; \)

\( \text{else append( } L, B\text{.first()} \) ; \( B = \text{rest}(B) \) ; \)

\( \text{append( } L, A) ; // all remaining elements of } A \)

\( \text{append( } L, B) ; // all remaining elements of } B \)

\( \text{return } L \)
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.

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

Quicksort an \( n \)-element array:
1. Divide: Partition the array into two subarrays around a pivot \( x \) such that elements in lower subarray \( \leq x \) elements in upper subarray.
2. Conquer: Recursively sort the two subarrays.

Key: Linear-time partitioning subroutine.
Pseudocode for quicksort

\[
\text{QUICKSORT}(A, p, r) \\
\text{if } p < r \\
\quad \text{then } q \leftarrow \text{PARTITION}(A, p, r) \\
\quad \text{QUICKSORT}(A, p, q-1) \\
\quad \text{QUICKSORT}(A, q+1, r) \\
\text{Initial call: } \text{QUICKSORT}(A, 1, n)
\]

Partitioning subroutine

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

Invariant:

\[
\begin{array}{cccccc}
 & p & \leq x & \quad i & \geq x & ? \\
\end{array}
\]

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

Wikipedia / “video”

Worst-case of quicksort

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

\[
T(n) = T(0) + T(n-1) + \Theta(n) \\
= \Theta(1) + T(n-1) + \Theta(n) \\
= T(n-1) + \Theta(n) \\
= \Theta(n^2) \quad \text{(arithmetic series)}
\]

Best-case analysis (For intuition only!)

If we’re lucky, \(\text{PARTITION}\) splits the array evenly:

\[
T(n) = 2T(n/2) + \Theta(n) \\
= \Theta(n \log n) \quad \text{(same as merge sort)}
\]

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

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

What is the solution to this recurrence?
More intuition

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

\[ I(n) = 2U(n/2) + \Theta(n) \quad \text{lucky} \]
\[ U(n) = I(n-1) + \Theta(n) \quad \text{unlucky} \]

Solving:

\[ I(n) = 2(I(n/2 - 1) + \Theta(n/2)) + \Theta(n) \]
\[ = 2I(n/2 - 1) + \Theta(n) \]
\[ = \Theta(n \lg n) \quad \text{Lucky!} \]

How can we make sure we are usually lucky?

The master method

The master method applies to recurrences of the form

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

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

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:** $T(n) = \Theta(n^{\log_b a})$.

2. $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.
   **Solution:** $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Examples

**EX.** $T(n) = 4T(n/2) + n$

- $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$.
  **CASE 1:** $f(n) = O((n^2 - \epsilon))$ for $\epsilon = 1$.
  \[ \therefore T(n) = \Theta(n^2) \]

**EX.** $T(n) = 4T(n/2) + n^2$

- $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$.
  **CASE 2:** $f(n) = \Theta(n^2 \log^{k+1} n)$, that is, $k = 0$.
  \[ \therefore T(n) = \Theta(n^2 \log n) \]

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 $\epsilon = 1$
  and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.
  \[ \therefore T(n) = \Theta(n^3) \]

**EX.** $T(n) = 4T(n/2) + n^3 \log n$

- $a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3 \log n$.
  Master method does not apply. In particular, for every constant $\epsilon > 0$, we have $n^\epsilon = \omega(\log n)$.

Three common cases (cont.)

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

3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$.
   - $f(n)$ grows polynomially 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 $\epsilon = 1$
  and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.
  \[ \therefore T(n) = \Theta(n^3) \]

Idea of master theorem

**Recursion tree:**

\[
\begin{array}{c}
\text{Recurrence tree:} \\
\quad f(n) \\
\quad a \\
\quad f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b) \\
\quad a \\
\quad f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2) \\
\quad \vdots \\
\quad T(1)
\end{array}
\]
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.

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

Randomized quicksort analysis

Let $T(n)$ be 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 

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

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

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

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

Take expectations of both sides.

Calculating expectation

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

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

Linearity of expectation.
Calculating expectation

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

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

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

Calculating expectation

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

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

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

Linearity of expectation; \( E[X_k] = 1/n \).

Calculating expectation

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

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

**Prove:** \( E[T(n)] \leq an \lg n \) 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:** \( \sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{2} n^2 \) (exercise).

Substitution method

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

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

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

\[ \leq an \lg n, \]

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

Alternative materials

- Quicksort average case analysis
  
  http://eid.ee/10z
  
  https://coursetxt.mit.edu/6.046/home/lectures/quickpages/av120z.html

- http://eid.ee/10y - MIT Open Courseware - Asymptotic notation, Recurrences, Substitution Master Method
21.2.2013

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.

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

- Is that the best we can do?
- Remember: using comparisons $<$, $>$, $\leq$, $\geq$ we cannot 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:** $A[1..n]$, where $A[i] \in \{1, 2, ..., k\}$.
- **Output:** $B[1..n]$, sorted.
- **Auxiliary storage:** $C[1..k]$.

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

```plaintext
A:  4  1  3  4  3
B:  □ □ □ □ □
C:  0  0  0  0

for i ← 1 to k
    do C[i] ← 0
```
Loop 2

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
\end{array}
\quad \quad
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
C: & 1 & 0 & 2 & 2 \\
\end{array}
\]

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

September 28, 2005  Copyright © 2005 J. Erik D. Demaine and Charles E. Leiserson  1.5.19

Loop 3

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & & & \\
\end{array}
\quad \quad
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
C: & 1 & 0 & 2 & 2 \\
\end{array}
\]

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

September 28, 2005  Copyright © 2005 J. Erik D. Demaine and Charles E. Leiserson  1.5.22

Loop 4

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
A: & 4 & 1 & 3 & 4 & 3 \\
B: & & & 3 & 4 & \\
\end{array}
\quad \quad
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
C: & 1 & 1 & 2 & 5 \\
C': & 1 & 1 & 2 & 4 \\
\end{array}
\]

for \( j \leftarrow n \) downto \( 1 \)
\[
\text{do } B[C[A[j]]] \leftarrow A[j] \\
\text{\( C[A[j]] \leftarrow C[A[j]] - 1 \)}
\]

September 28, 2005  Copyright © 2005 J. Erik D. Demaine and Charles E. Leiserson  1.5.24

Analysis

\[
\begin{align*}
\Theta(k) & \quad \{ \text{for } i \leftarrow 1 \text{ to } k \\
& \quad \text{do } C[i] \leftarrow 0 \\
\Theta(n) & \quad \{ \text{for } j \leftarrow 1 \text{ to } n \\
& \quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
\Theta(k) & \quad \{ \text{for } i \leftarrow 2 \text{ to } k \\
& \quad \text{do } C[i] \leftarrow C[i] + C[i-1] \\
\Theta(n) & \quad \{ \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{align*}
\]

September 28, 2005  Copyright © 2005 J. Erik D. Demaine and Charles E. Leiserson  1.5.28

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!

September 28, 2005  Copyright © 2005 J. Erik D. Demaine and Charles E. Leiserson  1.5.20

Stable sorting

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

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

Exercise: What other sorts have this property?
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.

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 \( \Rightarrow \) 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

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

\( r = 8 \Rightarrow b/r = 4 \) 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?**
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 $r$ 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 $T(n, b) = \Theta(bn/\lg n)$.

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?

0101100 0101100 0101100 0101100
1001010 0010010 0010010 0010010
1111000 1111000 0101000 0101000
1001001 1001001 1001001 1001001
0101000 0101000 1111000 1111000
0010000 0010000 1001010 1001010

• 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 sort left to right sort

```
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; /* mask of the already sorted area */

  array_end:
    mask = mask | curbit;
    /* area under mask is now sorted */
    curbit >>= 1; /* next bit */

  for( ; (i < size) && (! (ARRAY[i] & curbit)); i++)
    goto array_end;
  if( i >= size )
    goto array_end;

  group = ARRAY[i] & mask;
  /* save current prefix snapshot */

  for( ; ; )
  {
    if( ++i >= size )
      goto array_end;
    if( (ARRAY[i] & mask) != group )
      goto new_mask;
    /* new prefix */
    if( ! (ARRAY[i] & curbit) )
    {
      tmp = ARRAY[i];
      ARRAY[i] = ARRAY[j];
      ARRAY[j] = tmp;
      j += 1;
    }
  }

  new_mask:

  if( (curbit & curbit) )
    /* bit is 0 — need to swap with previous location of 1, if any */
    tmp = ARRAY[i];
    ARRAY[i] = ARRAY[j];
    ARRAY[j] = tmp;
    j = i; /* swap and increment to the next possible 1 */
  }
```

Jaak Vilo, Univ. of Tartu

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/

- 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-Titanium 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

Year 2008 Results

Year 2009 Results

Sort Benchmark

- http://sortbenchmark.org/
- Sort Benchmark Home Page
- We have a new benchmark called new GraySort, new 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 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 $A[(n+1)/2]$ or $A[(n+1)/2]$ \\

**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 $A[p]$

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

$k = q - p + 1$ // nr of elements in subarr

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

**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 – expected $O(n)$
- Find i’th value: worst case $O(n)$ – see CLRS
Lists: Array

- 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

Linked lists

- Array indexed from 0 to $n - 1$:
  - $k = 1$: $O(1)$
  - $1 < k < n$: $O(k)$
  - $k = n$: $O(n)$
- Insert before or after the $k$th element:
  - $k = 1$: $O(n)$
  - $1 < k < n$: $O(1)$
  - $k = n$: $O(n)$
- Delete the $k$th element:
  - $k = 1$: $O(n)$
  - $1 < k < n$: $O(n)$
  - $k = n$: $O(n)$

- Singly-linked list with head and tail pointers:
- Doubley linked list:

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 cannot be used...
- Or can it?

Skip List

A skip list, introduced by Pagh (Pagh 1999), is a randomised balanced tree data structure organized as a tower of increasingly sparse linked lists. Level $i$ of a skip list is a linked 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 $d$-level skip list, each node stores $d$ pointer pointers and a successor pointer for each list on which it appears, for an average of $d^2$ pointers per node. The lists at the highest levels act as “super lists” that allow the sequence of nodes to be traversed rapidly. 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 $\log_{\frac{1}{p}}$ nodes are searched on average per level, giving an average search time of $O\left(\log_{\frac{1}{p}} n\right)$ with a node at level 0. Skip lists have been extensively studied (Pagh 1999; Pagh et al. 1998; Berson 1995; Kienhöfer and Puschmann 1996; Kienhöfer et al. 1993), and because they require no global balancing operations are particularly useful in parallel systems (Galambos et al. 1998; Galambos and Mossé 1997).
Skip lists

- Build several lists at different “skip” steps
- O(n) list
- Level 1: ~ n/2
- Level 2: ~ n/4
- ...
- Level log n ~ 2-3 elements...

Skip List

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

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

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 y, we compare x with y = key(y)
    - x > y: we return element(y)
    - x < y: we “scan forward”
    - x = y: we “drop down”
  - If we try to drop down past the bottom list, we return NO_SUCH_KEY
• Example: search for 78

What is a Skip List

• A skip list for a set S of distinct (key, element) items is a series of lists S₀, S₁, .., Sₖ such that
  - Each list Sᵢ contains the special keys ±∞ and ±∞
  - List Sᵢ contains the keys of S in nondecreasing order
  - Each list is a subsequence of the previous one, i.e.,
    S₀ ⊆ S₁ ⊆ .. ⊆ Sₖ
  - List Sₖ contains only the two special keys
• We show how to use a skip list to implement the dictionary ADT

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

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 y, we compare x with y = key(y)
    - x > y: we return element(y)
    - x = y: we “scan forward”
    - x < y: we “drop down”
  - If we try to drop down past the bottom list, we return NO_SUCH_KEY
• Example: search for 78

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

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 y, we compare x with y = key(y)
    - x > y: we return element(y)
    - x = y: we “scan forward”
    - x < y: we “drop down”
  - If we try to drop down past the bottom list, we return NO_SUCH_KEY
• Example: search for 78
Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution.
- It contains statements of the type
  
  \[ b = \text{random}() \]
  
  - if \( b = 0 \)
  
  - else \( (b = 1) \)
  
- Its running time depends on the outcomes of the coin tosses.

- We analyze the expected running time of a randomized algorithm under the following assumptions:
  - the coins are unbiased, and
  - the coin tosses are independent.

- The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads").

- We use a randomized algorithm to insert items into a skip list.

- It contains statements of the type
  
  \[ \text{if } b = 0 \]
  
  - do \( A \)
  
  - else \( (b = 1) \)
  
  - do \( B \) …

- We remove all but one list containing only the two special keys.

- Example: remove key 34

Deletion

- To remove an item with key \( x \) from a skip list, we proceed as follows:
  - We 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 \( p_i \) is in list \( S_i \).
  - We remove positions \( p_0, p_1, \ldots, p_i \) 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

Insertion

- To insert an item \((x, e)\) 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 \( i \) is \( b \), we add to the skip list new lists \( S_{i+1}, \ldots, S_{i+k} \), each containing only the two special keys.
  - We search for \( x \) in the skip 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 \( j = 0, \ldots, i \), we insert item \((x, e)\) into list \( S_j \), after position \( p_j \).

- Example: insert key 15, with \( i = 2 \)

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 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 \( 1/2^i \)
  - Fact 2: If each of \( m \) items is present in a set with probability \( p \), the expected size of the set is \( mp \)

- Consider a skip list with \( m \) items
  - By Fact 1, we insert an item in list \( S_i \) with probability \( 1/2^i \)
  - By Fact 2, the expected size of list \( S_i \) is \( m/2^i \)

- The expected number of nodes used by the skip list is

\[
\sum_{i=0}^{\infty} \frac{m}{2^i} = \frac{m}{2-1} < 2m
\]

- Thus, the expected space usage of a skip list with \( m \) items is \( O(m) \)
21.2.2013

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 $np$.

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 yet another probabilistic fact:
  
  **Fact 4:** The expected number of coin tosses required in order to get tails is 2

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