Conclusions

• Algorithm complexity deals with the behavior in the long-term
  – worst case
  – average case
  – best case

• In practice, long-term sometimes not necessary
  – E.g. for sorting 20 elements, you don’t need fancy algorithms...

Physical ordered list ~ array

• Memory /address/
  – Garbage collection

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

• Disk
  – Disk fragmentation

Big-Oh notation classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Informal</th>
<th>Intuition</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(n) ∈ o (g(n))</td>
<td>f is dominated by g</td>
<td>Strictly below</td>
<td>&lt;</td>
</tr>
<tr>
<td>f(n) ∈ O(g(n))</td>
<td>Bounded from above</td>
<td>Upper bound</td>
<td>≥</td>
</tr>
<tr>
<td>f(n) ∈ Θ(g(n))</td>
<td>&quot;equal to&quot;</td>
<td>&quot;equal to&quot;</td>
<td>=</td>
</tr>
<tr>
<td>f(n) ∈ ω(g(n))</td>
<td>Bounded from below</td>
<td>Lower bound</td>
<td>≤</td>
</tr>
</tbody>
</table>

Linear, sequential, ordered, list ...

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

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...

```
1 2 3
4 5 6
7 8 9
```

2D array

```
\[ A[i][j] = A + (i*nr\_el\_in\_row + j)*el\_size \]
```

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 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

Abstract Data Types (ADT)

- Some common ADTs, which have proved useful in a great variety of applications, are
  - Container
  - Stack
  - List
  - Graph
  - Set
  - Queue
  - Multiset
  - Priority queue
  - Map
  - Double-ended queue
  - Multimap
  - Double-ended priority queue
- Each of these ADTs may be defined in many ways and variants, not necessarily equivalent.

Abstract data types:

- Dictionary
- 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
  - B-tree
  - B+ tree
  - Bx tree
  - B-sharp tree
  - B* tree
  - B++ tree

- **Balanced trees**
  - Binary search trees
  - AVL trees
  - Red-black trees
  - 2-3-4 trees
  - Treaps

- **Hashing**
  - Hashing
  - Hash maps
  - Hash sets

- **Trie**
  - Radix trees
  - Tries

### Hashes, Graphs, Other

- **Hashes**
  - Bloom filter
  - Distributed hash table

- **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

### Lists: Array

- **Size**: 
  - Array (memory address)
  - *array* [size MAX_SIZE]

<table>
<thead>
<tr>
<th>size</th>
<th>MAX_SIZE</th>
</tr>
</thead>
</table>
  | 0 1  | 5 6 7 5 2 | #

- **Operations**
  - Insert 8 after L[2]
  - Delete last

### Lists: Array

- **Size**: 
  - Array (memory address)
  - *array* [size MAX_SIZE]

<table>
<thead>
<tr>
<th>size</th>
<th>MAX_SIZE</th>
</tr>
</thead>
</table>
  | 0 1  | 5 6 7 5 2 | #

- **Operations**
  - Insert 8 after L[2]
  - 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

Linked lists: add

Linked lists: delete (+ garbage collection?)

Operations

- Array indexed from 0 to n – 1:

<table>
<thead>
<tr>
<th>Operation</th>
<th>O(1)</th>
<th>O(1)</th>
<th>O(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the kth element</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert before or after the kth element</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete the kth element</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Singly-linked list with head and tail pointers

<table>
<thead>
<tr>
<th>Operation</th>
<th>O(1)</th>
<th>O(1)</th>
<th>O(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the kth element</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert before or after the kth element</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete the kth element</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* under the assumption we have a pointer to the kth node, O(1) otherwise
Improving Run-Time Efficiency

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

  **Singly-linked list:**
  \[
  \text{list}_\text{head} \quad \text{list}_\text{tail}
  \]

  **Doubly-linked list:**
  \[
  \text{list}_\text{head} \quad \text{list}_\text{tail}
  \]

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

Improving Efficiency

**Singly-linked list:**

- Access/change the kth element: \( O(n) \) if \( k \neq 1 \)
- Insert before or after the kth element: \( O(n) \)
- Delete the kth element: \( O(n) \)

**Doubly-linked list:**

- Access/change the kth element: \( O(1) \)
- Insert before or after the kth element: \( O(1) \)
- Delete the kth element: \( O(1) \)

Introduction to linked lists

- Consider the following struct definition

  ```c
  struct node {
      string word;
      int num;
      node *next;
  }
  ```

  ```c
  node *p = new node;
  ```

  - How can you add another node that is pointed by \( p->\text{link} \)?

  ```c
  node *q;
  ```

  ```c
  p->next = NULL;
  ```
Introduction to linked lists

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";

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:**
- **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

Multiple lists, single free list

Hack: allocate more arrays ...

```
use integer division and mod
AA[(i-1)/7] => [ (i-1)% 7 ]
LIST(10) = AA[1][2]
LIST(10) = 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)

<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
<th>MAX_SIZE-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

First = List[F]  Pop_first : { return List[F++]; }  Last = List[L-1]  Pop_last : { return 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.
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 ever be?**

![Stack visualization](image)

- When full, allocate bigger table dynamically, and copy all previous values there
- **O(n)** ?

---

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

```cpp
BinarySearch(A[0..N-1], value, low, high)
{
  if (high < low)
    return -1 // not found
  mid = (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)
  { // Note: not (low + high) / 2 !!
    mid = low + (high - low) / 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 => A[18] ? <
• x => A[9] ? >
• x => 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...

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 \( \leq 2^h \) leaves. Thus, \( n! \leq 2^h \).

\[ h \geq \log(n!) \]

\[ \geq \log((n/e)^n) \]

\[ = n \log n - n \log e \]

\[ = \Omega(n \log n). \]

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

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

  \[ = \Omega(n \log n) \]

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

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

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

\[ \geq \frac{n}{2} \log_2 n/2 \]

\[ = \Omega(n \log n). \]
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

Merge-Sort(A, p, r)
if p<r
then q = (p+r)/2 // floor
Merge-Sort(A, p, q )
Merge-Sort(A, q+1, r)
Merge(A, p, q, r)

It was invented by John von Neumann in 1945.

Example

• Applying the merge sort algorithm:

```
L = new list; // empty
while(A not empty and B not empty)
    if A.first() <= B.first()
        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:

$$T(n) = \begin{cases} 
\Theta(1) & n = 1 \\
2T(\frac{n}{2}) + \Theta(n) & n > 1 
\end{cases}$$
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/

\[ a \quad b \]
\[ L[a] \leq L[b] \]
\[ a \quad b \]
\[ L[a] > L[b] \]

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 \( \leq x \) \( \leq \) elements in upper subarray.

2. **Conquer:** Recursively sort the two subarrays.
3. **Combine:** Trivial.

**Key:** Linear-time partitioning subroutine.

Pseudocode for quicksort

```
QUICKSORT(A, p, r)
if p < r
then q ← PARTITION(A, p, r)
QUICKSORT(A, p, q−1)
QUICKSORT(A, q+1, r)
```

**Initial call:** QUICKSORT(A, 1, n)
Partitioning subroutine

\[ \text{PARTITION}(A, p, q) \] > \[ A[p..q] \]

\[ i \leftarrow A[p] \]    \[ \text{pivot} = A[p] \]

\[ j \leftarrow p \]

for \[ j \leftarrow p + 1 \] to \[ q \]

do if \[ A[j] \leq x \]

then \[ i \leftarrow i + 1 \]

exchange \[ A[i] \leftrightarrow A[j] \]

return \( i \)

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

Partitioning version 2

pivot = \( A[R] \);

\( i = L; j = R - 1; \)

while \( i < j \)

while \( A[i] < \text{pivot} \) \( i++; \) // will stop at pivot latest

while \( i < j \) and \( A[j] \geq \text{pivot} \) \( j--; \)

if \( i < j \) { swap( \( A[i], A[j] \) ); \( i++; j--; \}

\( A[R] = A[i]; \)

\( A[i] = \text{pivot}; \)

return \( i \);

Wikipedia / “video”

Best-case analysis

(For intuition only!)

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

\[ T(n) = 2T(n/2) + \Theta(n) \]

\[ = \Theta(n \lg n) \] (same as merge sort)

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

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

What is the solution to this recurrence?

Analysis of “almost-best” case

\[ T(\frac{1}{10} n) \quad T(\frac{9}{10} n) \]
Choice of pivot in Quicksort

• Select median of three ...

• Select random – opponent can not choose the winning strategy against you!

More intuition

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

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

Solving:

L(n) = 2(L(n/2 - 1) + Θ(n/2)) + Θ(n)
    = 2L(n/2 - 1) + Θ(n)
    = Θ(n lg n)

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.

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)
Bentley–McIlroy 3-way partitioning

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

Swap equals to center after partition

less    equal    greater

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  
Jon 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

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

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

Randomized quicksort analysis (continued)

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

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

Analysis (continued)

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

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

Linearity of expectation.

Calculating expectation

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

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

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

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

Calculating expectation

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

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

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

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

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

Calculating expectation

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

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

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

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

Summations have identical terms.

Hairy recurrence

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

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

\[ E[T(n)] \leq 2 \sum_{k=2}^{n-1} \frac{a^k \log k + \Theta(n)}{n} \]
\[ = 2a \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n) \]
\[ = an \log n - \left( \frac{an}{4} - \Theta(n) \right) \]
\[ \leq an \log n, \]

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

Alternative materials

- Quicksort average case analysis
  
  http://eid.ee/10z
  
  https://secweb.cs.odu.edu/~zeil/cs361/web/website/lectures/quick/pages/ar01s05.html
- http://eid.ee/10y - MIT Open Courseware - Asymptotic notation, Recurrences, Substitution Master Method

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. \( f(n) = O(n^{\log_b a - \varepsilon}) \) for some constant \( \varepsilon > 0 \).
   
   - \( f(n) \) grows polynomially slower than \( n^{\log_b a} \) (by an \( n^\varepsilon \) factor).
   
   Solution: \( T(n) = \Theta(n^{\log_b a}) \).
Three common cases

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

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

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

Three common cases (cont.)

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

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

Examples

**Ex.** \( T(n) = 4T(n/2) + n \)

\( a = 4, \ b = 2 \Rightarrow n^{\log_b a} = n^2; \ f(n) = n \).
**Case 1:** \( f(n) = O(n^{\log_b a - \varepsilon}) \) for \( \varepsilon = 1 \).
\( \therefore T(n) = \Theta(n^2) \).

**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:** \( f(n) = \Omega(n^{\log_b a + k}) \), that is, \( k = 0 \).
\( \therefore T(n) = \Theta(n^{\log \log n}) \).

**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:** \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \) for \( \varepsilon = 1 \)
and \( 4(n/2)^3 \leq cn^3 \) (reg. cond.) for \( c = 1/2 \).
\( \therefore T(n) = \Theta(n^3) \).

**Ex.** \( T(n) = 4T(n/2) + n^2 \log n \)

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

Examples (cont.)

**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:** \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \) for \( \varepsilon = 1 \)
and \( 4(n/2)^3 \leq cn^3 \) (reg. cond.) for \( c = 1/2 \).
\( \therefore T(n) = \Theta(n^3) \).

**Ex.** \( T(n) = 4T(n/2) + n^2 \log n \)

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

Idea of master theorem

**Recursion tree:**

\[ \begin{align*}
  f(n) & \quad a \\
  f(n/b) & \quad a \\
  \vdots & \\
  f(n/b^k) & \quad \vdots \\
  \vdots & \\
  T(1) & \\
\end{align*} \]

\( f(n) \) \( f(n/b) \) \( f(n/b^2) \) \( T(1) \)
Idea of master theorem

**Recursion tree:**

\[
\begin{array}{c}
\text{CASE 1: The weight increases} \\
\text{geometrically from the root to the} \\
\text{leaves. The leaves hold a constant} \\
\text{fraction of the total weight.}
\end{array}
\]

\[
\begin{array}{c}
\text{CASE 2: (} h = 0 \text{) The weight} \\
\text{is approximately the same on} \\
\text{each of the } \log_n n \text{ levels.}
\end{array}
\]

\[
\begin{array}{c}
\text{CASE 3: The weight decreases} \\
\text{geometrically from the root to the} \\
\text{leaves. The root holds a constant} \\
\text{fraction of the total weight.}
\end{array}
\]

Back to sorting
We can sort in $O(n \log n)$

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

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

for $i \leftarrow 1$ to $k$
  do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $n$
  do $C[A[j]] \leftarrow C[A[j]] + 1$  
  \[ C[i] = \text{\{key = }i\text{\}} \]
for $i \leftarrow 2$ to $k$
  do $C[i] \leftarrow C[i] + C[i-1]$  
  \[ C[i] = \text{\{key }\leq i\text{\}} \]
for $j \leftarrow n$ downto 1
  do $B[C[A[j]]] \leftarrow A[j]$
  
  $C[A[j]] \leftarrow C[A[j]] - 1$

**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>
<th></th>
</tr>
</thead>
</table>

for $i \leftarrow 1$ to $k$
  do $C[i] \leftarrow 0$

**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>
<th></th>
</tr>
</thead>
</table>

for $i \leftarrow 1$ to $n$
  do $C[A[j]] \leftarrow C[A[j]] + 1$  
  \[ C[i] = \text{\{key = }i\text{\}} \]
Loop 3

\[
\begin{align*}
A: & \quad \begin{array}{|c|c|c|c|c|}
\hline & 1 & 2 & 3 & 4 & 5 \\
\hline
B: & & & & & \\
\hline
\end{array} \\
C: & \quad \begin{array}{|c|c|c|c|}
\hline & 1 & 0 & 2 & 2 \\
\hline
C': & 1 & 1 & 3 & 5 \\
\end{array}
\end{align*}
\]

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

Loop 4

\[
\begin{align*}
A: & \quad \begin{array}{|c|c|c|c|c|}
\hline & 1 & 2 & 3 & 4 & 5 \\
\hline
C: & \quad \begin{array}{|c|c|c|c|}
\hline & 1 & 1 & 2 & 5 \\
\hline
C': & 1 & 1 & 2 & 4 \\
\end{array}
\end{array} \\
B: & \quad \begin{array}{|c|c|c|}
\hline & 3 & 4 \\
\hline
\end{array}
\end{align*}
\]

for \( j \leftarrow n \) downto \( 1 \)
\[
\begin{align*}
do & \quad B[C[A[j]]] \leftarrow A[j] \\
& \quad C[A[j]] \leftarrow C[A[j]] - 1
\end{align*}
\]

Analysis

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

\(\Theta(n + k)\)

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.

\[
\begin{align*}
A: & \quad \begin{array}{|c|c|c|c|c|}
\hline & 4 & 1 & 3 & 4 & 3 \\
\hline
B: & 1 & 3 & 3 & 4 & 4 \\
\end{array} \\
\end{align*}
\]

Exercise: What other sorts have this property?

Radix sort

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

Correctness of radix sort (continued)

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

| 8 | 8 | 8 | 8 |

\( r = 8 \Rightarrow b/r = 4 \) 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. Moreover, 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 $r = \log n$ implies $T(n, b) = \Theta(bn/\log n)$.

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

**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>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>rose</td>
<td>rose</td>
<td>rose</td>
<td>rose</td>
</tr>
<tr>
<td>ash</td>
<td>ash</td>
<td>ash</td>
<td>ash</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
</tbody>
</table>

**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 $\lceil \log 2000 \rceil = 11$ passes.

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

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>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
</tbody>
</table>

**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>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>red</td>
<td>red</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>red</td>
<td>red</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

/* 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, rebits;
    register SORTTYPE mask, curbit, group;
    rebits = sizeof(SORTTYPE) * 8;
    curbit = 1 << (rebits - 1);
    while(curbit)
    {
        ARRAY = 0;
        new_mask: for(;
            (i < size) && (! (ARRAY[i] & curbit));
            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(;
            1 << curbit | ARRAY[i] != 0;
            i++);
        /* Reached end of array */
        if(ARRAY[i] & mask) != group)
            goto new_mask;
        /* New prefix */
        if(! (ARRAY[i] & curbit))
        { /* bit is 0 – need to swap with previous location of 1, A[i] = A[j] */
            tmp = ARRAY[i];
            ARRAY[i] = ARRAY[j];
            ARRAY[j] = tmp;
        } /* Swap and increase to the next possible 1 */
    }
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

Bitwise from left to right

0010000
0010010
0101000
0101100
10010
1
0
10010
0
1
1001001
1111000

• Swap '0' with first '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

0.98
0.17
0.09
0.26
0.72
0.94
0.68
0.78
0.23
0.66
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-Ianium 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

- 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

- 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 3 hour (90s is a minimum reliability requirement).
  - 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

\[
\begin{align*}
\text{Minimum}(A) \\
1 & \text{ min } = A[1] \\
2 & \text{ for } i = 2 \text{ to length}(A) \\
3 & \text{ if } \text{min} > A[i] \text{ then } \text{min} = A[i] \\
4 & \text{ return min} \\
\end{align*}
\]

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 \( O(n) \)

\[
\begin{align*}
\text{Randomised-select( A, p, r, i )} \\
\text{if } p=r \text{ then return } A[p] \\
q = \text{Randomised-Partition}(A,p,r) \\
k = q - p + 1 \quad // \text{nr of elements in subarr} \\
\text{if } i < k \text{ then return } \text{Randomised-Partition}(A,p,q,i) \\
\text{else return } \text{Randomised-Partition}(A,q+1,r,i-k) \\
\end{align*}
\]
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 Pugh (Pugh, 1990), is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists. Level $L$ of a skip list is a linked list of all nodes in increasing order for keys. Each such link is in level $L+1$. It appears in level $L$ 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 $\frac{1}{p}$ pointers per node. The lists at the highest level act as "super lists" that allow the separation 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 $\frac{1}{p}$ nodes are searched on average per level, giving an average search time of $O(\log \frac{n}{\log n})$ with $n$ nodes at level 0. Skip lists have been extensively studied (Pugh, 1990; Pagh et al., 1996; Datar et al., 2002; Kornfelder and Prediger, 1996; Kornfelder et al., 1999), and because they require no global balancing operations are particularly useful in parallel systems (Zaharia et al., 1996; Gehrke and Monnig, 1997).

Skip lists

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

Fig. 1. A skip list with $n=8$ nodes and $\log_2 n=3$ levels.
Skip Lists

typedef struct nodeStructure *node;
typedef struct nodeStructure{
    valueType 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 S₀, S₁, …, Sₜ such that
  – Each list Sᵢ contains the special keys +∞ and −∞.
  – List S₀ contains the keys of S in non-decreasing order
  – Each list is a subsequence of the previous one, i.e.,
    S₀ ≺ S₁ ≺ … ≺ Sₜ
  – List Sₜ contains only the two special keys
• We show how to use a skip list to implement the dictionary ADT

Randomized Algorithms

• A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
  • It contains statements of the type
    \( b = \text{random}(k) \)
    \( \text{if } b = 0 \)
    \( \text{do } A \ldots \) 
    \( \text{else if } b = 1 \) 
    \( \text{do } B \ldots \) 
  • 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”)
Insertion

- To insert an item \((x, a)\) 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 \(i < b\), we add the item \((x, a)\) to the skip list new lists \(S_0, S_1, \ldots, S_b\) 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 \(S_0, S_1, \ldots, S_b\).
  - For \(j = 0, 1, \ldots, i\), we insert \((x, a)\) into list \(S_j\) after position \(p_j\).

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

```
\[
\begin{align*}
S_0 & : x = 10, a = 10 \\
S_1 & : x = 15, a = 15 \\
S_2 & : x = 20, a = 20 \\
\end{align*}
\]
```

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 \(p_j\) is in list \(S_j\).
  - We remove positions \(p_0, p_1, \ldots, p_i\) of the item 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

```
\[
\begin{align*}
S_0 & : x = 10, a = 10 \\
S_1 & : x = 15, a = 15 \\
S_2 & : x = 20, a = 20 \\
\end{align*}
\]
```

Implementation v2

- 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
  - Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them

```
\[
\begin{align*}
S & : x = 10, a = 10 \\
S & : x = 15, a = 15 \\
S & : x = 20, a = 20 \\
\end{align*}
\]
```

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: If each of \(w\) items has height \(O(\log n)\)
  Fact 2: If each of \(w\) items is present in a set with probability \(p\), the expected size of the set is \(\mathbb{E}(w)\)
- Consider a skip list with \(w\) 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 \(2^{-i}\)
- By picking \(i = \mathbb{E}(\log n)\), we have that the probability that \(S_{\mathbb{E}(\log n)}\) has at least one item is at most \(2^{-i}\)
- Thus a skip list with \(w\) items has height at most \(3\log n\) with probability at least \(1 - 1/w^2\)
- Consider a skip list with \(w\) items
  - By Fact 1, we insert an item in list \(S_i\) with probability \(1/2^i\)
  - By Fact 2, the probability that list \(S_i\) has at least one item is at most \(2^{-i}\)
  - Thus a skip list with \(w\) items has height at most \(3\log w\) with probability at least \(1 - 1/w^2\)

```
\[
\begin{align*}
S & : x = 10, a = 10 \\
S & : x = 15, a = 15 \\
S & : x = 20, a = 20 \\
\end{align*}
\]
```

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
- 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.