CS 3/5

www.cs.uky.edu/~raphael/courses/
CS315.html

A        B          C
Mr.  Raphael   Finkel
Dr.  Rack   Goldstein
Prof.  $0, 50$

email list
help@cs.uky.edu

Basic building blocks —
Lists + Trees

examples of data structures
way to represent information
so it can be manipulated
packaged along with routines to manipulate
Abstract Data Type (ADT)

API (Application Program Interface) hides the internal details

Tool:

\[
\begin{align*}
\text{use} & \quad \text{specification (API)} \\
\text{implementation} & \\
\end{align*}
\]

Linked List of Data operations (API)

create empty list
delete list

\[\begin{align*}
\mathcal{O}(1) & \quad \text{insert data at front of list} \\
\mathcal{O}(1) & \quad \text{delete data at front} \\
\mathcal{O}(n) & \quad \text{insert data at end} \\
\mathcal{O}(n) & \quad \text{delete data at end} \\
\mathcal{O}(n) & \quad \text{count length} \\
\mathcal{O}(n) & \quad \text{search for particular data} \\
\mathcal{O}(\log n) & \quad \text{sort data}
\end{align*}\]

"big-O of ..."

"order of ..."
Linked List of Integer: Nodes

handle = header

1) build a node A, with value inside
2) A.next = handle.next
3) handle.next = A

Efficiency (optimization)
1) if fast enough, leave it alone.
2) Can you wait a year?
3) Is your algorithm right?
4) Determine where the time is spent.

gcc -O3
header

```
front
rear
pseudo count
```

```
3
1
4
1
```

empty list

```
front
rear
pseudo count
```

```
1
```

complexity of an algorithm:

\[ \mathcal{O}(...)]

Queues, stacks, dequesues

```
① → black box ② ←
```

Queue: 0, 2, 3, 4

Stack: ①, ②

push, pop

Dequeue: 0, 2, 3, 4
Stack of integers

operations: makeEmptyStack(), bool isEmptyStack(s), int popStack(s), void pushStack(s x s, int data)

Implementation 1: Linked list of integers

makeEmptyStack: makeEmptyList
isEmptyStack: isEmptyList
popStack: deleteFromList
pushStack: insertFromList

Implementation 2: Array of integers

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

push(3)
push(1)
push(4)
push: array[count] <- data
count += 1
pop: count -= 1
return array[count]
Queue of integer

\[ q \leftarrow \text{make Empty Queue} \]
\[ \text{bool} \leftarrow \text{is Empty Queue}(q) \]
\[ \text{void insert Queue}(*q, \text{int data}) \]
\[ \text{int} \leftarrow \text{delete Queue}(*q) \]

**Implementation 1: Linked List**

![Linked List Diagram]

---

% randGen.pl | cards = ---
% cards ---

```
#include <stdio.h>

int main(int argc, char *argv[])
```

```c
fscanf(stdin, "%d", &nextInt);
```
Implementation 2 of queues

array

\[
\begin{array}{ccccccc}
3 & 1 & 4 & 1 & 5 & 9 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
f \\
\end{array}
\]

Deque of integer,
Implementation 2: array

similar to Queue

need "retreat( )" function

Implementation 1: doubly-linked list
Searching / Sorting

API: n data elements (int)

data structures D

insert (int data, D)

bool search (int data, D)

Representation 1: Linked list

insert: \(\Theta(1)\) (at front)

search: \(\Theta(n)\) pseudo-data: target of search

Rep 2: Sorted linked list

insert: \(\Theta(n)\) pseudo-data: \(\infty\)

search: \(\Theta(n)\)

Rep 3: Array

\[\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}\]

counter

insert: \(\Theta(1)\)

search: \(\Theta(n)\)
Rep 4: Sorted array

insert: $O(n)$
search: $O(\log n)$ Binary Search

Counter

32 49 56 83 92

Insert at end,

32 49 56 83 92

Look from start

$\text{start}$
Recursion theorem

Given \( C_n = f(n) + a \cdot C_{n/b} \)

where \( f(n) = \Theta(n^k) \),

Then

<table>
<thead>
<tr>
<th>( a &lt; b^k )</th>
<th>( \Theta(n^k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b^k )</td>
<td>( \Theta(n^k \log n) )</td>
</tr>
<tr>
<td>( a &gt; b^k )</td>
<td>( \Theta(n^{\log_b a}) )</td>
</tr>
</tbody>
</table>

Binary search:

\( C_n = 1 + C_{n/2} \)

\( f(n) = 1 \quad k = 0 \)

\( a = 1 \)

\( b = 2 \)

\( \Rightarrow C_n = \Theta(n^k / \log n) = \Theta(\log n) \)
\( \Theta(f(n)) = \text{no worse than } f(n) \)
\( = \text{at most } f(n) \)

\( \Omega(f(n)) = \text{no better than } f(n) \)
\( = \text{at least } f(n) \)

\( \Theta(f(n)) = \text{exactly } f(n) \)

Data structure: sorted binary tree

![Binary Tree Diagram]

Balanced: all paths have about the same length
Path: from root down to a leaf
exact balance:
#nodes: $2^n - 1$

balanced:
path lengths are with 1 of each other

"vine"
worst tree

---

<table>
<thead>
<tr>
<th>num</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>$2^n - 1$</td>
<td>$n$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\log_2 x$</td>
</tr>
</tbody>
</table>

insertion into a balanced tree of $n$ nodes is $O(\log n)$

searching in a balanced tree is $O(\log n)$

building a vine: $O(n^2)$

building a balanced tree: $O(n \log n)$
Trees: operations

2. Build
   1. Insert into
      offline / online

- one value at a time
- have all values in advance
- build a balanced tree
  in \( O(n \log n) \)

3. Search (exact, close)

4. Traversals

```c
void symmetric(node)
{
    if tree is empty
        return;
    else
        node->left = symmetric(left child);
        visit(node);
        symmetric(node->right);
}
```
```c
void preorder (node) {
    if node is empty {
    } else {
        visit (node);
        preorder (node -> left);
        preorder (node -> right);
    } // not empty
    } // preorder()

void postorder (node) {
    if node is empty {
    } else {
        postorder (node -> left);
        postorder (node -> right);
        visit (node);
    } // not empty
    } // postorder()
```

Representation 6: Hashing (scatter storage)

later,
insert: O(1)
search: O(1)
Finding the $j$th element in a set of numbers.

1. Easy situation: numbers are sorted.
   - want highest. $a[100]$ $\Theta(1)$
   - want 3rd highest. $a[98]$ $\Theta(1)$
   - want median. $a[50]$ $\Theta(1)$

2. Numbers not sorted.
   - want highest. $\Theta(n)$

   \[
   \text{largest} = -\infty
   \]
   
   \[
   \text{foreach value in array} \exists
   \]
   
   \[
   \text{if } (\text{value} > \text{largest}) \exists
   \]
   
   \[
   \text{largest} = \text{value} \notag
   \]
   
   \[
   \text{else if } (\text{value} \geq \text{nextLargest}) \exists
   \]
   
   \[
   \text{nextLargest} = \text{largest} \notag
   \]
   
   \[
   \text{largest} = \text{value} \notag
   \]
   
   \[
   \text{else} \notag
   \]
   
   \[
   \text{nextLargest} = \text{value} \notag
   \]

3. 
6. not sorted, want j-th largest.
   \( \Theta(j \cdot n) \): keep an array of the j largest so far.

7. median (unsorted) \( \Theta(n^2) \)
   better: \( \Theta(n) \)
   based on partitioning (half-hearted sorting) on a pivot.

\[
\begin{array}{c}
\text{arbitrary} \\
\uparrow \\
\text{small} \quad | \quad \text{large} \\
\uparrow \\
X \quad X \\
\uparrow \\
\quad | \quad P_2 \\
\uparrow \\
\quad | \quad P \\
\uparrow \\
\end{array}
\]

before

after 1 partitioning

after 2

Complexity:

recurrence relation

\[ C_n = n + C_{n/2} \]

\[ a = 1 \quad b^k = 2^k = 2 \]

\[ b = 2 \quad \Theta(n^k) = \Theta(n) \]

\( k = 1 \)
\( \Theta(1) \) constant
\( \Theta(\log n) \) logarithmic
\( \Theta(n) \) linear
\( \Theta(n^2) \) quadratic
\( \Theta(n^3) \) cubic
\( \Theta(2^n) \) exponential

Nico Lomuto's algorithm
"comb" method.

\[
\begin{align*}
\text{pivot value} &= \text{array}[0] \\
P &= 0 \\
\text{for } i &
\end{align*}
\]

\[
\begin{array}{cc}
\text{small} & \text{big} \\
\uparrow & \uparrow \\
P & c \rightarrow
\end{array}
\]
Sorting methods

Insertion Sort:

\[ \Theta(n^2) \]

Before: [5 2 1 7 9 0 3]

After: [3 2 1 0 5 7 9]

sorted
Selection sort

\[ \Omega(n^2) \]

QuickSort (C. A. R. Hoare)

initial

partition

after 1 step

recurse on both sides
Initially: $O(n)$

$log n$ steps each costing $O(n)$ for partitioning

$\Rightarrow O(n \log n)$

Suggestions for improvement:
1) Choose median of 3 or 5 as pivot
2) Don't recurse if size is < 10

After all recursion is done, run 1 pass of insertion sort.

Recursion theorem

\[
C_n = f(n) + a \cdot C_{n/b}^{\text{partitioning}} + \\
\begin{cases} 
1 & \text{if } k=1 \\
2(n/2) & \text{if } k=2 
\end{cases}
\]

$\delta = \Theta(n^k \log n) = \Theta(n \log n)$
Consider a worse partitioning

\[ C_n = n^k + 2C_{n/2} \]

\[ k = 1 \]
\[ a = 2 \]
\[ b = 3/2 \]

\[ \Theta \left( \frac{1}{n^{\log_3 2}} \right) \]

\[ \Theta \left( \frac{\log_3 (2)}{n^{2/3}} \right) = \Theta \left( n^{1.7} \right) \]

Shell Sort (Donald Shell 1959)

Each pass uses a span value s.

For each offset 0 \ldots s-1
Next pass: smaller span.

offset 0

\[
\begin{array}{c}
\cdots \\
\uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \\
5 \\
\cdots \\
\end{array}
\]

offset 1

\vdots

last pass: span = 1

Heaps

\[
\begin{array}{c}
42 \\
3 \\
\end{array} \rightarrow
\begin{array}{c}
3 \\
42 \quad 43 \\
\end{array} \rightarrow
\begin{array}{c}
2 \\
3 \\
42 \quad 43 \\
\end{array} \rightarrow
\begin{array}{c}
2 \\
3 \\
42 \quad 43 \\
\end{array} \rightarrow
\begin{array}{c}
2 \\
3 \\
42 \quad 43 \\
\end{array}
\]

Sift up: swap with a parent, may be repeated.

Insertion: \( \log n \)
delete from heap always deletes smallest element.

Sift down: swap with the better child, maybe repeatedly.
Uses of a heap.

1) Priority queue
2) Sorting heap sort
   - Put numbers in an array.
   - Make the array a heap.
     top-heavy: largest value at root
     \[ \Theta(n \log n) \]
   - Repeatedly delete from heap.
     \[ \Theta(n \log n) \]

Fast heapification

\[ n \left \lceil \frac{1}{2} \sum_{j=1}^{i} \left( \left \lceil \frac{1}{2^j} \right \rceil \right) \right \rceil \approx \frac{n}{2} \sum_{j=2}^{i} \frac{1}{2^j} \]
Bin Sort

Know the range of key
Know no duplicates

Array of bits

Space: $O(r)$ where $r$ is the size of range
place in value: $O(n)$
read the bits in order: $O(r)$

total time: $O(n+r)$

Radix sort on digits (decimal), $n$ numbers
100 200 369 427 150 062 042 746

Cost: j passes

n inserts
100 062 945 427 369
200 052
150

$O(j \cdot n)$

j is no bigger than log n
$O(n \cdot \log n)$

042 100 200 369 427 946
\[ C_n = n + \frac{1}{2} C_{n/2} \]

\[ n \leq k \]

θ(\(n^k \log n\)) = θ(\(n \log n\)) ^

1) guaranteed.
2) stable

Stable: preserves order of duplicate keys.

Trees: online, mostly balanced trees

AVL trees: Adelson Velskii Landes

Red-black trees: Guibas, Sedgewick 1978

Every node is either red or black

Root is black

Red nodes have only black children.

All paths have same number of black nodes

Height (worst path) is ≤ 2log n

Rotation: y

\[ \begin{array}{c}
  x \quad \text{right} \quad y \\
  a \\
  b \\
  c
  \end{array} \]

\[ \begin{array}{c}
  x \quad \text{left} \quad b \\
  y \\
  c
  \end{array} \]
to insert value v:
place it in its location in binary tree.
color it red.
walk up the tree from v to root, fixing
as needed.
color root black

case 1: parent and uncle are red.
circled: black

\[
\begin{align*}
&\quad \text{9} \\
p &\quad \text{u} \quad \Rightarrow \quad \text{recolor} \quad \text{g}^* \\
v &\quad \text{u}
\end{align*}
\]

case 2: parent red, uncle black, v inside

\[
\begin{align*}
&\quad \text{9} \\
&\quad \text{u} \quad \Rightarrow \quad \text{rotate} \quad \text{g}^* \\
&\quad \text{u} \quad \text{p} \quad \text{v} \quad \text{up} \quad \text{p} \quad \text{down} \\
&\quad \text{1} \quad \text{2} \quad \text{3}
\end{align*}
\]

case 3: parent red, uncle black v outside

\[
\begin{align*}
&\quad \text{9} \\
&\quad \text{u} \quad \Rightarrow \quad \text{recolor} \quad \text{g} \\
&\quad \text{p} \quad \text{v} \quad \text{up} \quad \text{p} \quad \text{down} \quad \text{g} \quad \text{g}^*
\end{align*}
\]
Insert 1, 2, 3, 4, 5, 6

1. \(\text{insert} \ 2\)
2. \(\text{insert} \ 3\)
3. \(\text{recolor} \ 3\)
4. \(\text{rotate} \ 3\)
5. \(\text{recolor} \ 4\)
6. \(\text{rotate} \ 4\)
7. \(\text{recolor} \ 5\)
8. \(\text{insert} \ 6\)

Case 1

Case 2

Case 3
Insert 5, 2, 7, 4, 3, 1

- Insert 5
- Insert 2
- Insert 7
- Insert 4
- Insert 3

Case 1

Case 2

Case 3

Case 3

Case 1
Binary trees

Depth: best $\log_2 n$
worst $n$
random expected $O(\log n)$
balanced (AVL or red-black): $2 \log_2 n$
Insertion: $O(\log n)$ if not worst

Traversals: pre-order
post-order
symmetric order (in-order)

Deletion

Leaves are easy.
Internal nodes $d$.
1) Mark "deleted" but leave in tree
2) Replace it with a near neighbor
   if $d$ has one child, put in $d$'s place
   if $d$ has 2 children, $d$'s successor
   $s = RL^*$. put $s$ in $d$'s place
   if $s$ has no left child, but a right
2) Replace

\[ d \text{ has 2 children} \]
\[ S = d \cdot R \cdot L \]
if \( S \) has a right child \( C \),
move \( C \) in place of \( S \).

\[ \Rightarrow \text{not easy.} \]

---

Extensions to binary trees

Ternary tree

```
42 50
/    \
7 1 45 48 70 90
/ \
20 21 \\
42
```