CS 315

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

A       B       C
Mr.     Raphael  Finkel
Dr.     Rack     Goldstein
Prof.  jac      

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:
- use specification (API)
- implementation

Linked List of Data
operations (API)
  - create empty list
  - delete list

\( O(1) \) insert data at front of list
\( O(1) \) delete data at front
\( O(n) \) insert data at end
\( O(n) \) delete data at end
\( O(n) \) count length
\( O(n) \) search for particular data
\( O(n \log n) \) sort data

"big-O of . . ."
"order of . . ."
Linked List of Integer: Nodes

handle = header

- empty list
- insert at front

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
1

empty list

front
rear
pseudo
count

0

complexity of an algorithm = \( \Theta(...) \)

Queues, stacks, dequesues

Queue: 1, 2, 3, 4
Stack: 1, 2
push, pop
Deque: 0, 2, 3, 5
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: insertFrontList

Implementation 2: Array of integers

```
3 1 4 0 0 0 0
```
MAXSIZE = 9

push(3)  
push(1)  
push(4)  
push: array[count] <- data  
count += 1

pop: count -= 1  
return array[count]
Queue of integer

```c
q ← make Empty Queue ()
bool ← is Empty Queue (q)
void insert Queue (*q, & int data)
int ← delete Queue (*q)
```

**Implementation 1: Linked List**

![Linked List Diagram]

```c
% randGen.pL  |  cards
% cards
```

```c
#include <stdio.h>

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

```
Implementation 2 of queues

Array:

```
array
3 1 4 1 5 9 2 3
```

Dequeuing 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

Rep 2: Sorted linked list

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

search: \( \Theta(n) \)

Rep 3: Array

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

counter

insert: \( \Theta(1) \)

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

counter

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

32 49 56 83 92

70 70 92

70 83

32 49 56 83 92

"""

70 83 92

inset at end,
look from 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/b )</th>
<th>( C_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a &lt; b^k )</td>
<td>( \Theta(n^k) )</td>
</tr>
<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:

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

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

\( a = 1 \)

\( b = 2 \)

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

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

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

Data structure: sorted binary tree

Root: 42
Internal node: 45
Leaf: 2

Balanced: all paths have about the same length.
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 \( \Theta(n \log n) \)

(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);
    }
}
```

```c
void postorder(node)
{
    if node is empty
    {
    }
    else
    {
        postorder(node->left);
        postorder(node->right);
        visit(node);
    }
}
```

Representation 6: Hashing (scatter storage)
Later,
inset: O(1)
search: O(1)
Finding the jth element in a set of numbers.

1. Easy situation: numbers are sorted.
   want highest. \( a[100] \) \( \Theta(1) \)

2. want 3rd highest. \( a[98] \) \( \Theta(1) \)

3. want median. \( a[50] \) \( \Theta(1) \)

4. numbers not sorted.
   want highest. \( \Theta(n) \)

\[
\text{largest} := -\infty
\]

\[
\text{foreach value in array } \}
\]
\[
\text{if (value} > \text{largest) } \}
\]
\[
\text{largest} := \text{value};
\]
\[
\}
\]

5. not sorted, want 2nd highest.

\[
\text{largest} = \text{nextLargest} := -\infty
\] \( \Theta(n) \)

\[
\text{foreach value in array } \}
\]
\[
\text{if (value} > \text{largest) } \}
\]
\[
\text{nextLargest} := \text{largest};
\]
\[
\text{largest} := \text{value};
\]
\[
\text{else if (value} > \text{nextLargest) } \}
\]
\[
\text{nextLargest} := \text{value};
\]
\[
\}
\]
not sorted, want j'th largest.
\( O(j \cdot n) \): keep an array of the j largest so far.

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

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)
\( \Theta(1) \) 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.

\[ \text{pivot value} = \text{array}[0] \]
\[ p = 0 \]
for \( c \)

\[ \text{small} \quad \text{big} \]
\[ p \quad c \rightarrow \]
Final step

2 1 0 3 1 4 17 6 9 10
Sorting methods

Insertion Sort: \[ \Theta(n^2) \]

Before

| 4 | 2 | 1 | 7 | 9 | 0 | 3 |

After

| 2 | 1 | 0 | 9 | 7 | 3 | 4 |

| 2 | 1 | 0 | 3 | 7 | 9 | 4 |
Selection sort

```
<table>
<thead>
<tr>
<th>sorted</th>
<th>unsorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/1/</td>
<td></td>
</tr>
</tbody>
</table>
```

in place

```
<table>
<thead>
<tr>
<th>small</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

smallest

```
| 1/1/1/ |
```

$O(n^2)$

Quick Sort (C. A. R. Hoare)

```
<table>
<thead>
<tr>
<th>initial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

partition

```
<table>
<thead>
<tr>
<th>low</th>
</tr>
</thead>
</table>
```

| high     |

```
|         |
```

after 1 step

```
<table>
<thead>
<tr>
<th>recur on both sides</th>
</tr>
</thead>
</table>
```
Initially: $\Theta(n)$

$log n$ steps
each costing $\Theta(n)$
for partitioning

$\Rightarrow \Theta(n \log n)$

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

$$C_n = f(n) + a \frac{C_{n/b}}{\text{partition}}$$

$$C_{n/2} + 2C_{n/2}$$

$$k = 1$$
$$\alpha = 2$$
$$b = 2$$
$$b^k = 2$$

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

\[ C_n = n^k + 2 \sum_{m=\lfloor n/2 \rfloor}^{n/3} 2a/3 \]

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

\[ \Theta \left( \frac{\log n^{\frac{2}{3}} 2^k}{n} \right) \]

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

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

\[
\begin{array}{ccc}
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\end{array}
\]

offset 1

last pass: span = 1

Heaps

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

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

\[ \text{heap} \rightarrow \text{random} \]
\[ \Theta(n \log n) \]
- Repeatedly delete from heap.

\[ \text{heap} \rightarrow \text{sorted} \]
\[ \Theta(n \log n) \]

Fast heapification

\[ n < \sum_{j=0}^{\log n} \frac{1}{2^j} \Rightarrow n \approx \frac{n}{2} \cdot \sum_{j=0}^{\log n} \frac{1}{2^j} \]
**Bin Sort**

Know the range of key  
Know no duplicates

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>79</th>
<th>array of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>↑</td>
<td>↑</td>
<td>100</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td>79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>79</th>
<th>array of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>79</td>
</tr>
</tbody>
</table>

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

total time: $\Theta(m+r)$

**Radix sort**  
digits (decimal), $n$ numbers

| 100  | 200 | 369 | 427 | 150 | 062 | 042 | 746 |

Cost: $j$ passes

<table>
<thead>
<tr>
<th>n</th>
<th>100</th>
<th>200</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>062</td>
<td>042</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$O(jn)$

$j$ is no bigger than $\log n$  
$O(\log n)$
\[ C_n = n + \binom{n}{2} \]

\[ n \leq k \]

\[ \Theta(n^k \log n) = \Theta(n \log n) \]

(1) guaranteed.

(2) stable

Stable: preserves order of duplicate keys.

Trees: online, mostly balanced trees

AVL trees: Adelson-Velskii Landis

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 \( \leq \log n \)

Rotation: \( y \) right \( x \) left
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

\[
\begin{array}{c}
\circ \quad 9 \\
\circ \quad u \\
\circ \quad p \\
\circ \quad n \\
\circ \quad 3 \\
\circ \quad 1 \\
\circ \quad 2 \\
\end{array}
\]

\[
\begin{array}{c}
\circ \quad p \quad \rightarrow \quad 9^+ \\
\circ \quad u \quad \rightarrow \quad \circ \quad p \\
\circ \quad u \quad \rightarrow \quad \circ \quad p \\
\circ \quad n \quad \rightarrow \quad \circ \quad p \\
\circ \quad 3 \quad \rightarrow \quad \circ \quad p \\
\circ \quad 1 \quad \rightarrow \quad \circ \quad p \\
\circ \quad 2 \quad \rightarrow \quad \circ \quad p \\
\end{array}
\]

case 2: parent red, uncle black, \( v \) inside

\[
\begin{array}{c}
\circ \quad 9 \\
\circ \quad c \\
\circ \quad p \\
\circ \quad u \\
\circ \quad n \\
\circ \quad 3 \\
\circ \quad 1 \\
\circ \quad 2 \\
\end{array}
\]

\[
\begin{array}{c}
\circ \quad p \quad \rightarrow \quad u \quad \uparrow \quad \text{rotate} \\
\circ \quad u \quad \rightarrow \quad v \quad \downarrow \quad \text{rotate} \quad p \quad \downarrow \quad \text{rotate} \\
\circ \quad n \quad \rightarrow \quad u \quad \downarrow \quad \text{rotate} \quad p \quad \downarrow \quad \text{rotate} \\
\circ \quad 3 \quad \rightarrow \quad c \quad \downarrow \quad \text{rotate} \quad p \quad \downarrow \quad \text{rotate} \\
\circ \quad 1 \quad \rightarrow \quad c \quad \downarrow \quad \text{rotate} \quad p \quad \downarrow \quad \text{rotate} \\
\circ \quad 2 \quad \rightarrow \quad c \quad \downarrow \quad \text{rotate} \quad p \quad \downarrow \quad \text{rotate} \\
\end{array}
\]

case 3: parent red, uncle black \( v \) outside

\[
\begin{array}{c}
\circ \quad 9 \\
\circ \quad c \\
\circ \quad p \\
\circ \quad u \\
\circ \quad n \\
\circ \quad 3 \\
\circ \quad 1 \\
\circ \quad 2 \\
\end{array}
\]

\[
\begin{array}{c}
\circ \quad c \quad \rightarrow \quad 9 \\
\circ \quad p \quad \rightarrow \quad u \\
\circ \quad u \quad \rightarrow \quad p \quad \uparrow \quad \text{rotate} \\
\circ \quad n \quad \rightarrow \quad p \quad \downarrow \quad \text{rotate} \\
\circ \quad 3 \quad \rightarrow \quad c \quad \downarrow \quad \text{rotate} \\
\circ \quad 1 \quad \rightarrow \quad c \quad \downarrow \quad \text{rotate} \\
\circ \quad 2 \quad \rightarrow \quad c \quad \downarrow \quad \text{rotate} \\
\end{array}
\]
Insert 1, 2, 3, 4, 5, 6

1
   \rightarrow
  /  \                      \rightarrow
\text{insert}_2  \text{insert}_3   \text{3 recolor}  \rightarrow  2
   \rightarrow
  /  \                      3 rotate
\text{insert}_2  \text{insert}_4  \rightarrow
   \rightarrow
  /  \                      \rightarrow
\text{insert}_4  \text{insert}_5   2
   \rightarrow
  /  \                      3 rotate
\text{insert}_5  \text{insert}_6  \rightarrow
   \rightarrow
  /  \                      \rightarrow
\text{case 3 recolor}  \text{case 3 rotate}  2
   \rightarrow
  /  \                      \rightarrow
\text{1 3 4}  \text{1 3}  \text{1 4}
   \rightarrow
  /  \                      \rightarrow
\text{4 5}  \text{3 5}  \text{4 5 6}
Insert 5, 2, 7, 4, 3, 1

\[ \text{insert } 5 \rightarrow 5 \xrightarrow{\text{insert } 2} 5 \xrightarrow{\text{insert } 7} 2 \xrightarrow{\text{insert } 4} 5 \xrightarrow{\text{insert } 3} 6 \]

\[ \text{case 1: recombine} \]

\[ \text{case 2:} \]

\[ \text{case 3: negative color} \]

\[ \text{rotate} \]

\[ \text{case 3} \]

\[ \text{insert } 1 \rightarrow 3 \xrightarrow{\text{insert } 7} 3 \xrightarrow{\text{insert } 1} 3 \xrightarrow{\text{insert } 1} 5 \]

\[ \text{case 1} \]
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 $c$ 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

![Ternary Tree Diagram]

\[
\frac{\log_3 n}{\frac{1}{3} \log_3 n} \quad A
\]

\[
\frac{\log_2 n}{\log_2 n} \quad B
\]

\[ A/B = 1.05 \]
Quad trees

Every internal node has a discriminant $\langle x, y \rangle$ that describes the children.

Modification: stop subdividing when there are only a few nodes (bucket size $b^2 \leq 10$).
Leaf can hold up to 10 values "buckets."

Searching requires $O(\log n)$ steps down the tree, then $O(b)$ match operations.

Nearest neighbor of $P$:
1) Find bucket where $p$ would be
2) Try nearby buckets if necessary,
   shrinking the ball determined by nearest neighbor so far.

Generalization: 3-d Octrees
K-d trees:

for high-dimension points

Recursively:
1) if # points is small (≤ b), make a bucket.
2) Find the dimension with greatest range.
3) For that dimension, find median value.
4) Subdivide problem into two subproblems, based on discriminant: (dim, median)
5) recurse

Use: OCR (optical character recognition)

Insertion into K-d tree is easy: add to right bucket, subdividing if it overflows.
2-3 trees.
\[ C_n = 1 + 3 c^{\frac{2}{3}} n \]

\[ a = 3 \]
\[ b = \frac{3}{2} \]
\[ b^k = 1 \]
\[ k = 0 \]

\[ \Theta \left( n^{\log_b \frac{a}{b}} \right) \]
\[ = \Theta \left( n^{\log_{\frac{3}{2}} 3} \right) \]
\[ \approx \Theta \left( n^{2.71} \right) \]

Check first if sorted

\[ C_n = n + 3 c^{\frac{2}{3}} n \]
3, 1, 4, 1, 5, 9, 2, 6, 5, 3

1) Insert online into binary tree, ties to left, pre/postorder traversal.

2) Insert online top-light heap, breadth-first traversal.
3) Place in an array, then heapify.
   Breadth-order traversal.

4) Place in the ternary tree (online).
   Pre-order traversal.

5) Place in an array; complete 5 steps of selection sort.
6) place in array, do 5 steps of insertion sort.

\[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 2, 1, 3, 4, 1, 2, 3, 4\]

7) insert online into 2-3 tree

\[1, 3, 4 \rightarrow 1, 1, 2, 4, 5, 9 \rightarrow 1, 1, 2, 4, 5\]

\[3, 5, 1 \rightarrow 1, 3, 5 \rightarrow 1, 3, 5, 1, 3, 5\]

\[1, 1, 2, 4, 9, 1, 2, 4, 9, 1, 2, 3, 4, 5, 6, 9\]