CS 315

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

A B C
Mr. Raphael Finkel
Dr. Rack Goldstein
Prof. S & C

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)
hide the internal details

Tool:
\[ \text{use specification (API)} \]
\[ \text{implementatio} \]

Linked List of Data Operations (API)
create empty list
delete list

\[ \Theta(1) \quad \text{insert data at front of list} \]
\[ \Theta(1) \quad \text{delete data at front} \]
\[ \Theta(n) \quad \text{insert data at end} \]
\[ \Theta(n) \quad \text{delete data at end} \]
\[ \Theta(n) \quad \text{count length} \]
\[ \Theta(n) \quad \text{search for particular data} \]
\[ \Theta(n \log n) \quad \text{sort data} \]

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

handle = header

```
data  next
3     1  4
```

empty list

handle

insert at front

5

1) build a node A, with value inside
2) A.next = handle.next
3) A = 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:
\[ \Theta(\ldots) \]

Queues, stacks, dequeues

1. \( \rightarrow \) black box \( \rightarrow \) 3. insert front
2. \( \leftarrow \) insert rear

Queue: 0, 1, 2, 4
Stack: 1, 2
push, pop
Deque: 0, 2, 3, 1
Stack of integers

Operations:
- `makeEmptyStack()`
- `bool isEmptyStack(S)`
- `int popStack(S)`
- `void pushStack(S, x, int data)`

Implementation 1: Linked list of integers

- `makeEmptyStack`: `makeEmptyList`
- `isEmptyStack`: `isEmptyList`
- `popStack`: `deleteFromList`
- `pushStack`: `insertFrontList`

Implementation 2: Array of integers

```
| 3 | 1 | 4 | |
```

MaxSize: 3

- `push(3)`
- `push(1)`
- `push(4)`

```
push: array[count] <- data
count += 1

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

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]

dummy node

echo "% randGen.pl | cards ....

stdout
stdin
stderr

#include <stdio.h>

+scanf(stdin, "%d", &nextInt);

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

insert 3

insert 1

delete

delete
Implementation 2 of queues

array

[3 4 1 4 1 7 9 2 3]

Dequeing of integer.

Implementation 2: array

similar to Queue

need "retreat()" function

Implementation 1: doubly-linked list

Diagram of doubly-linked list with a dummy node.
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

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\end{array}
\]

counter

insert: \( \Theta(1) \)

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

0

\text{counter}

\text{insert: } \Theta(n)

\text{search: } \Theta(\log n) \text{ Binary Search}

\text{insert at end,}

\begin{array}{c|c|c|c|c|c|c}
& 32 & 49 & 56 & 83 & 92 & 70 \\
\hline
\text{start}
\end{array}

\text{look from start}

\begin{array}{c|c|c|c|c|c|c}
& 32 & 49 & 56 & 83 & 92 & 70 & 83 & 92 \\
\hline
\text{start} & \uparrow & \uparrow & \times & \uparrow & \times
\end{array}

\times \downarrow
Recursion theorem

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

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

Then

| \( a < b^k \) | \( \Theta(n^k) \) |
| \( a = b^k \) | \( \Theta(n^k \log n) \) |
| \( a > b^k \) | \( \Theta(n^{\log_b a}) \) |

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

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>$\approx n \log_2 X$</td>
</tr>
</tbody>
</table>

inserting 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

1. Insert into
   offline / online

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

2. Build

3. Search (exact, close)

4. Traversals

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

void postorder (node)
{
    if node is empty
    {
    }
    else
    {
        postorder (node \rightarrow left);
        postorder (node \rightarrow right);
        visit (node);
    }
    // not empty
}
// postorder()
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 > largest)} \}
\[\text{largest} = \text{value};\]
\[\}\]

\[\text{not sorted, want 2nd highest.}\]
\[\text{largest = nextLargest} = -\infty\]

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

(7) median (unsorted) \( O(n^2) \)
better: \( O(n) \)

based on partitioning (half-hearted sorting) on a pivot.

```
<table>
<thead>
<tr>
<th>arbitrary</th>
<th>before</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>p_1</td>
</tr>
<tr>
<td></td>
<td>large</td>
</tr>
</tbody>
</table>

X X X

```

after 1 partitioning

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

P

```

after 2

Complexity:

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

recurrence relation

\[
a = 1 \quad b^k = 2^k = 2 \quad \Theta(m^k) = \Theta(m)
\]

k = 1

b = 2

```
$O(1)$ constant
$O(\log n)$ logarithmic
$O(n)$ linear
$O(n^2)$ quadratic
$O(n^3)$ cubic
$O(2^n)$ exponential

Nico Lomuto's algorithm

"comb" method.

```
l : min index
p : pivot index

pivotVal = array[0]
p = 0
for c
    small big
    p c →
```
Sorting methods

Insertion Sort: \( \Theta(n^2) \)
Selectim sort

sorted  \quad unsorted
\begin{array}{c}
/ / / / /
\end{array}
small
in place
large
\begin{array}{c}
\vdash
\end{array}
\begin{array}{c}
smallest
\end{array}
before
\begin{array}{c}
/ / / / /
\end{array}

\Theta(n^2)

QuickSort (C. A. R. Hoare)

initial

partition

low

high

after 1 step

recuse on both sides
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

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

$\begin{align*}
\text{partition} & ; \quad k = 1 \\
\text{partition} & ; \quad a = 2 \\
n^1 + 2(C_{n/2}) & ; \quad b = 2 \\
\Theta(n^k \log n) & = \Theta(n/\log n)
\end{align*}$
Consider a worse partitioning

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

\[ k = 1 \]

\[ a = 2 \quad \text{compare } a : b^k \]

\[ b = 3/2 \quad 2 : 3/2 \]

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

**Shell Sort (Donald Shell 1959)**

Each pass uses a span value \( s \).

- **pass 1**

  - **subpass 1**

  - **subpass 2**

  - For each offset \( 0 \ldots 2^k - 1 \)
Next pass: smaller span.

Heaps

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.

Parent is at \([\text{index}/2]\)

Right child is at \([\text{index} \times 2 + 1]\)

Left child is at \([\text{index} \times 2]\)
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

\[
\begin{array}{c|c}
\text{heap} & \text{random} \\
\hline
\end{array}
\]
\[\uparrow\]
\[O(n \log n)\]
- Repeatedly delete from heap.

\[
\begin{array}{c|c}
\text{heap} & \text{sorted} \\
\hline
\end{array}
\]
\[\leftarrow \uparrow\]
\[O(n \log n)\]

Fast heapification

\[
\begin{array}{c|}
\text{heap} \\
\hline
\end{array}
\]
\[\uparrow\]

\[
m < n \sum_{2^j \leq n} \frac{1}{2^j} \leq n \sum_{2^j \leq n} \frac{1}{2^j} \]
Bin Sort

Know the range of key
Know no duplicates

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
100 200 369 427 150 062 042 946

Cost: $j$ passes
n inserts

$O(n)$

j is not bigger
than log n
$O(2 \log n)$
\[ C_m = n + \frac{1}{2} C_{n/2} \]

\[ n \leq k \leq b \]

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

1) guaranteed.
2) stable

Stable: preserves order of duplicate keys.

Trees: online, mostly balanced trees

AVL trees: Adelson-Velskii 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 \( \leq 2 \log n \)

Rotation:

\[ \begin{array}{c}
\text{left}
\end{array} \]

\[ \begin{array}{c}
\text{right}
\end{array} \]

\[ \begin{array}{c}
\text{left}
\end{array} \]

\[ \begin{array}{c}
\text{right}
\end{array} \]

\[ \begin{array}{c}
\text{left}
\end{array} \]

\[ \begin{array}{c}
\text{right}
\end{array} \]

\[ \begin{array}{c}
\text{left}
\end{array} \]
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.
circle: black

\[
\begin{align*}
\text{case 2: parent red, uncle black, } v \text{ inside} & \\
\text{case 3: parent red, uncle black } v \text{ outside} & \\
\end{align*}
\]

\[
\begin{align*}
&\text{case 2: parent red, uncle black, } v \text{ inside} \\
&\text{case 3: parent red, uncle black } v \text{ outside} \\
\end{align*}
\]
Insert 1, 2, 3, 4, 5, 6

1. Insert 2
2. Insert 3
3. Recolor
4. Insert 4
5. Insert 5
6. Case 3
7. Recolor
8. Insert 6
Insert 5, 2, 7, 4, 3, 1

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

Case 1: reco

Case 2: key color

Case 3: rotate

Case 3: rotate
Binary trees

Depth: best $\log_2 n$
worst $n$
random expected $\Theta(\log n)$
balanced (AVL or red-black): $2 \log_2 n$
Insertion: $\Theta(\log n)$ 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
   
   \[ s = d \cdot L \times R \]

   if \( s \) has a right child \( c \),
   move \( c \) in place of \( s \).

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

---

Extensions to binary trees
Ternary tree

\[
\begin{array}{c}
\text{cost of searching:}
\log_3 n \text{ levels}
\end{array}
\]

\[
\begin{array}{c}
5 \times 3 \text{ comparisons per level}
\end{array}
\]

\[
\begin{array}{c}
\left( \frac{5}{3} \log_3 n \right) A
\end{array}
\]

compare to binary tree

\[
\begin{array}{c}
\log_2 n \text{ levels}
\end{array}
\]

\[
\begin{array}{c}
1 \text{ comparison/level}
\end{array}
\]

\[
\begin{array}{c}
= \log_2 n B
\end{array}
\]

\[
\begin{array}{c}
A/B = 1.05
\end{array}
\]
Quad trees

Modification: stop subdividing when there are only a few nodes (bucket size b², 10) leaf can hold up to 10 values

"buckets"

Searching requires $O(\log n)$ steps down 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.
\[ C_n = 1 + 3 C_{\frac{2}{3} n} \]

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

\[ \Theta \left( n^{\log_b q} \right) \]

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

\[ \approx \Theta \left( n^{2.71} \right) \]

---

Check first if sorted

\[ C_n = n + 3 C_{\frac{2}{3} n} \]

\[ a = 3 \]
\[ b = \frac{3}{2} \]
\[ b^k = \frac{3}{2} \]
\[ k = 4 \]
1) Insert online into binary tree, ties to left, 
 prefix traversal.

pre:
3, 1, 2, 3, 4, 5, 9, 2, 6

post:
l, 2, 4, 3, 1, 5, 9, 6

2) Insert online top-light heap, 
breadth-first traversal.

l, 1, 3, 4, 2, 9, 5, 3, 6
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 \rightarrow 9, 2, 6, 5, 3

1, 3, 4

7) insert online into 2-3 tree

\[
\begin{align*}
&1, 3, 4 \rightarrow 1, 1_2, 4, 5, 9 \\
&1, 2, 4, 9 \rightarrow 1, 2, 4, 9 \rightarrow 1, 2, 4, 9
\end{align*}
\]

8) red-black
B trees (Ed McCreight)

like 2-3 trees

bucket size m where m-1 values, m indices fit in a block.

Example: block = 4KB index = 4B
value = 4B
m about = 512
Block

\[ \text{Insert: } \text{find the leaf} \]
\[ \text{Insert value in that leaf} \]
\[ \text{If leaf is now over-full, split it into 2 leaves,} \]
\[ \text{advancing middle value up 1 level. (Iterate up the tree)} \]

\[ \text{Usual state:} \]
\[ \text{Root is between 1 and } m-1 \text{ values.} \]
\[ \text{Other nodes: between } \frac{m-1}{2} \text{ and } m-1 \]

\[ \text{Height of tree } \mathcal{O}(\log n) \]

deletion: not pleasant.
\[ \text{From leaf: remove, maybe borrow} \]
\[ \text{From sibling} \]
\[ \text{From internal node: exchange with successor, then delete.} \]

B-trees are an example of using external storage for large data sets.
Hashing - a data structure for searching.

insert() : $O(1)$
search() : $O(1)$

Idea: find a value $k$ by looking in
an array at location $h(k)$

\[ D \]
\[
\begin{array}{c}
\text{ancient} \\
\text{ancient} \\
6 \\
5 \\
8 \\
9 \\
\end{array}
\]

It is an ancient mariner
6 7 1 2 6?

\[ \downarrow \]

collision.

Birthday paradox

\[ \Pr_{\text{no collision with } j \text{ people}} = \frac{365!}{(365-j)! 365^j} \]

if \( j \geq 23 \), then \( \Pr_{\text{no collision}} < \frac{1}{2} \)

\[ \implies \text{one must deal with collisions.} \]
Collision resolution

Open addressing: probe sequence.
if you don't find the target after one probe, try the next locations in sequence

linear probing
probe p is at \( h(k) + p \mod s \)
where \( s \) is size of array.
problems: clusters form
clusters coalesce
terrible behavior if almost full.
unacceptable behavior if '1/3 full.'

quadratic probing
probe p is at \( h(k) + p^2 \mod s \)
if two keys hash to same value,
their probe sequence is identical.
"secondary clustering"
add the hash rehash

probe p is at \( (p+1) h(k) \mod s \)
voids clustering.
h(k) must never be 0.