CS 315

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

A B C
Mr. Raphael Finkel
Dr. RaA Goldstein
Prof. Sa G

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)

implement

Linked List of Data Operations (API)

create empty list

delete list

\( \Theta(1) \) insert data at front of list

\( \Theta(1) \) delete data at front

\( \Theta(n) \) insert data at end

\( \Theta(n) \) delete data at end

\( \Theta(n) \) count length

\( \Theta(n) \) search for particular data

\( \Theta(n \log n) \) sort data

"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

empty list

front
rear
pseudo
count

0

complexity of an algorithm = \( O(\ldots) \)

Queues, stacks, dequesues

1 \rightarrow \text{black box} \rightarrow 3
2 \leftarrow 1

Queue: 1, 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, int data)

Implementation 1: Linked list of integers

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

Implementation 2: Array of integers

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
</table>

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

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

Implementation 1: Linked List

![Linked List Diagram]

\#include <stdio.h>

\%

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

```c
#include <stdio.h>

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

insert 3

insert 1

delete

delete
Implementation 2 of queues

array

Dequeue 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

Representation 2: Sorted linked list
insert: \( \Theta(n) \) pseudo-data: \( \infty \)
search: \( \Theta(n) \)

Representation 3: Array

\[
\begin{array}{cccccccc}
& & & & & & 0 & 1 & 2 \\
\hline
& & & & & & \text{counter} & & \\
\end{array}
\]

insert: \( \Theta(1) \)
search: \( \Theta(n) \)
Rep 4: Sorted array

0

\[ \begin{array}{cccccc}
0 & 1 & 2 & 3 & \cdots & \text{max} \\
\end{array} \]

counter

insert: \( O(n) \)

search: \( \Theta(\log n) \) Binary Search

insert at end,

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & 70 \\
\end{array}
\]

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & 70 & 83 & 92 \\
\end{array}
\]

\( \text{insert at start} \)

look from start

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & \text{X} \\
\end{array}
\]

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & \text{X} \\
\end{array}
\]

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & \text{X} \\
\end{array}
\]

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & \text{X} \\
\end{array}
\]

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & \text{X} \\
\end{array}
\]
Recursion theorem

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

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

Then

<table>
<thead>
<tr>
<th>When</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:

\[
C_n = 1 + C_{n/2}
\]

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

\( a = 1 \)

\( b = 2 \)

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

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

$Θ(f(n)) = \text{exactly } f(n)$

Data structure: sorted binary tree

![Binary Tree Diagram]

- Balanced: all paths from the root to a leaf 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>$\sim \log_2 x$</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
      \[\text{one value at a time}\]
      \[\text{have all values in advance}\]
      \[\Rightarrow \text{build a balanced tree}\]
      \[\in \mathcal{O}(n \log n)\]

3. Search (exact, close)

4. Traversals
   \[
   \text{symmetric (in-order)}
   \]
   \[
   \text{void symmetric(node)} \begin{cases}
   \text{if node is empty} \Rightarrow \\
   \text{else} \Rightarrow \\
   \text{node->left;}
   \text{visit (node);} \\
   \text{symmetric (node->right);} \\
   \end{cases}
   \]

```plaintext
// pre order (node)
if node is empty
else
    visit (node);
    preorder (node -> left);
    preorder (node -> right);
// not empty

// post order (node)
if node is empty
else
    postorder (node -> left);
    postorder (node -> right);
    visit (node);
// not empty

// post order ()
```

Representation 6: Hashing (scatter storage)
Later,
inset: \( O(1) \)
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. \( \Theta(n) \)

\[
largest = -\infty
\]

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

4. \[\text{not sorted, want 2nd highest.} \]
   - \[\text{largest = nextLargest = -\infty} \] \( \Theta(n) \)

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

\[ \text{median (unsorted) } \mathcal{O}(n^2) \]

better: \( \mathcal{O}(n) \)

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

\[
\begin{array}{c}
\text{arbitrary} \\
\uparrow \\
\text{small} \quad P_1 \quad \text{large} \\
\downarrow \quad \downarrow \quad \downarrow \\
\times \quad \times \\
\uparrow \\
\text{after 1 partitioning} \\
\text{after 2} \\
\end{array}
\]

Complexity:

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

Recurrence relation

\[
a = 1 \quad b^k = 2^k = 2^k \\
b = 2 \quad k = 1 \\
\mathcal{O}(n^k) = \mathcal{O}(n)\]
\(O(1)\) 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.

\[
l, r, \text{index, pivot index, } p \rightarrow \text{index, pivot index} \\
\]

pivot value = array[0]

\[
p \rightarrow 0 \\
\text{for } i \\
\]

small \[\rightarrow \text{big} \]
Sorting methods

Insertion Sort:

\[ \Theta(n^2) \]

\[
\begin{array}{c}
\text{before} \\
\text{sorted} \\
\text{after}
\end{array}
\]

\[
\begin{array}{c}
1 \quad 1 \quad 1 \\
1 \quad 1 \quad 1
\end{array}
\]
Select the set

\[
\begin{array}{c|c}
\text{sorted} & \text{unsorted} \\
\hline
1/1/1 & \\
\end{array}
\]

before

\[
\text{small in place} \uparrow \text{large} \uparrow \text{smaller}
\]

\[
1/1/1/1
\]

\[\Theta(n^2)\]

QuickSort (C.A.R. Hoare)

initial

\[
\text{partition}
\]

low high

\[
1/1/1/1
\]

after 1 step

recurse on both sides
Initially: \( \Theta(n) \)

\[ \log n \text{ steps} \]

\[ \text{each costing } \Theta(n) \]

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

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

\[ \begin{align*}
\text{partition} & \quad k = 1 \\
\alpha = 2 & \quad a = 2 \\
\frac{n}{b} & \quad b = 2 \\
\frac{n}{2} & \quad b^k = 2
\end{align*} \]

\[ \Theta(n^k \log n) = \Theta(n \log n) \]
Consider a worse partitioning
\[ C_n = n^k + 2 \frac{C_n}{1/3} \]

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

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

Shell Sort (Donald Shell 1959)

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

For each offset \( 0 \ldots 2^{\log s} - 1 \)
Next pass: smaller span.

\[
\begin{array}{c}
\vdots \\
\text{offset 0} \\
5 \\
\text{offset 1} \\
\vdots \\
\end{array}
\]

last pass: span = 1

Heaps

\[
\begin{array}{c}
42 \\
3 \\
\end{array} \
\Rightarrow 
\begin{array}{c}
42 \\
43 \\
\end{array} \
\Rightarrow 
\begin{array}{c}
2 \\
3 \\
43 \\
42 \\
90 \\
41 \\
\end{array}
\]

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

parent is at \[ \text{index}/2 \] 
left child is at \[ \text{index} \times 2 \]
right child is at \[ \text{index} \times 2 + 1 \]
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

\[ \text{heap} \quad \text{random} \]

\[ \Theta(n \log n) \]
- Repeatedly delete from heap.

\[ \text{sorted} \]

\[ \Theta(n \log n) \]

Fast heapification

Fast heapification
Bin Sort

Know the range of key
Know no duplicates

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

total time: $O(m+r)$

Radix sort by digits (decimal), $n$ numbers
100 200 369 427 150 062 042 146

Cost: $j$ passes

$n$ inserts

\[ O(j \cdot n) \]

$j$ is no bigger than $\log_{2} n$
\( C_n = n + \frac{1}{2} C_{n/2} \uparrow \)

\( n \leftarrow k \quad b \)

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

\( \text{guaranteed.} \)

\( \text{stable} \)

\text{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 \( \leq \log n \)

**Rotation:**

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

\( a \leftarrow b \quad c \leftrightarrow a \quad b \leftrightarrow c \)
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

\[ \text{case 1: parent and uncle are red.} \]
\[ \text{circle!black} \]

\[ \text{case 2: parent red, uncle black, v inside} \]

\[ \text{case 3: parent red, uncle black v outside} \]
Insert 1, 2, 3, 4, 5, 6

1 \xrightarrow{\text{insert}} 2 \xrightarrow{\text{insert}} 3 \xrightarrow{\text{insert}} 4 \xrightarrow{\text{insert}} 5 \xrightarrow{\text{insert}} 6

\text{case 3} \xrightarrow{\text{recolor}} \text{case 3} \xrightarrow{\text{rotate}} \text{case 1} \xrightarrow{\text{recolor}}
Insert 5, 2, 7, 4, 3, 1

\[\text{insert } 5 \rightarrow \boxed{5} \quad \text{insert } 2 \rightarrow \boxed{5} \quad \text{insert } 7 \rightarrow \boxed{5} \quad \text{insert } 4 \rightarrow \boxed{6} \]

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

\[\text{case 2: } \leftarrow \]

\[\text{case 3: } \downarrow \text{ ve color}\]

\[\text{case 3: } \rightarrow \text{ rotate}\]

\[\text{case 3: } \rightarrow \text{ rotate}\]

\[\text{insert } 3 \rightarrow \boxed{5} \quad \text{insert } 1 \rightarrow \boxed{5} \quad \text{insert } 1 \rightarrow \boxed{5}\]
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

\[ d \text{ has 2 children } \]

\[ S = d \cdot R \cdot L \cdot \star \]

if \( S \) has a right child \( C \),
move \( C \) in place of \( S \).

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

---

Extensions to binary trees

Ternary tree

\[
\begin{array}{c}
42,50 \\
\text{7,8} \\
\text{20,21} \\
\end{array}
\begin{array}{c}
45,48 \\
\text{7090} \\
\text{85,60} \\
\text{75,77} \\
\text{42} \\
\text{76}
\end{array}
\]

Cost of searching:

\[ \log_3 n \text{ levels} \]

573 comparisons per level

\[ \frac{2}{3} \log_3 n \text{ A} \]

Compare to binary tree

\[ \log_2 n \text{ levels} \]

1 comparison/level

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

\[ A/B = 1.05 \]
Quad trees

Every internal node has discriminant \((x, y)\) that describe 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

Internal node

Use: OCR (optical character recognition)

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

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

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

Check first if sorted

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

\[ a = 3 \]
\[ b = \frac{3}{2} \quad b^k = \frac{3}{2} \]
\[ k = 1 \]
3, 1, 4, 1, 5, 9, 2, 6, 5₂, 3₂

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

pre:
3₁ 1, l₂ 2 3₂
4 5₁ 5₂ 9 6

post:
l₂ 3₂ 2 l₁
5₂ 6 9 5₁ 4 3₁

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 2 3 2
1 3 1
1 3 1 2 3 4
```

7) insert online into 2-3 tree

```
1, 3, 4 → 1 1, 2 4 5 9 → 1 1, 2 4 5
```

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

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

8) red-black

```
1 4
```

```
3
```

```
1 4
```

```
3
```

```
1
```

```
3
```

```
1
```

```
3
```

```
1
```

```
4
```

```
1
```

```
4
```

```
3
```

```
1
```

```
4
```

```
1
```

```
4
```

```
3
```

```
1
```

```
4
```

```
1
```

```
4
```

```
3
```
B trees (Ed McCreight)

like 2-3 trees

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

Example: block = 4KB

value = 4B

m about = 512
Insert: find the leaf
    insert value in that leaf
    if leaf is now over-full,
        split it into 2 leaves,
        advancing middle value up 1 level.
    (iterate up the tree)

Usual state:
    root is between 1 and m-1 value.
    other nodes: between \( \frac{m-1}{2} \) and m-1

Height of tree \( O(\log n) \)

Deletion: not pleasant.
    from leaf: remove, maybe borrow
        from sibling
    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 \( h(k) \)

\[ h \text{ hash function.} \]

Birthday paradox

\[ \text{Prob( no collisions with } j \text{ people) } \]

\[ \frac{365!}{(365-j)! \cdot 365^j} \]

If \( j \geq 23 \), then \( \text{Prob} < \frac{1}{2} \)

\[ \implies \text{One must deal with collisions.} \]
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 \( \frac{1}{3} \) 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.
Online:

Zoom class: https://uky.zoom.us/j/ 739546678

telephone: 646-876-8923
passcode: 030135
office hours: 581609732
no passcode

open addressing
double hashing. $h_1(k), h_2(k)$
probe $p$ is at: $h_1(k) + p h_2(k)$
better alternative:
external chaining

$S \left[ \begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
\end{array} \right] \ 
\rightarrow \ \ \text{it} \ \rightarrow \ \text{insert into the chain}

\text{associated with } h(k) \ 
\text{make sure array is at least as big as}
\# \text{ of Keys} \ 
\text{average list length is } \frac{N}{n}$.
Optimizations

1) Always insert at front of list.
   Always re-position after search to front.
   locality of reference

2) For testing, use $h(c) \equiv 0$
   then use a good hash function

3) If you are worried about long lists,
   instead, use a binary tree, red-black tree,
   2-3 tree
   But it's usually better to increase $s$.

What is a good hash function?

- Fast.
- Uniform: equally likely to give any value 0...s-1
- Spreading: similar keys should hash far apart.

methods: Add all the chars of $K$ (mod $s$)
multiply some words with shifting.

Wisdom: it doesn't make much difference.
Extendible hashing

If $s$ is too small:

1) Relash everything with a bigger $s$.
   Causes a temporary "outage!"

2) Start with $s = 1$
   If chain is too long ($\geq 10$ elements)
   split chain based on last bit of $h(k)$.

   - even (last bit of $h$)
   - odd ("")
   - last bits 01
   - last bits 11

"Trie" $O(\log n)$

Scripting languages have hash tables built in.

Cryptographic hashes

Non-invertable: given $h(k)$, cannot derive $k$.
Purpose: store passwords.

- catching plagiarism.
- intrusion detection (tripwire)
- certificates on web