CS 315

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

A  B  C
Mr.  Raphael  Finkel
Dr.  RaC  Goldstein
Prof.  So So

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)
Application Program Interface (API) hides the internal details

Tool:

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

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 this 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

[Diagram of queue with nodes 3, 1, 4, 1]

empty list

([Diagram of empty list with front, rear, and pseudo count nodes set to 0])

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

Queues, stacks, dequeues

1. \rightarrow \text{black box} \rightarrow 2

Queue: \( \emptyset, 1, 2, 3, 4 \)

Stack: \( 1, 2 \)

push, pop

Deque: \( 0, 2, 3, 5 \)
Stack of integers

operations:  
  make Empty Stack(s)  
  bool is Empty Stack(s)  
  int pop Stack(s)  
  void push Stack(s, int data)

Implementation 1: Linked list of integers

make Empty Stack:  make Empty List  
is Empty Stack:  is Empty List  
pop Stack:  delete From List  
push Stack:  insert From List

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

Implementation 1: Linked List

stdout
stdin
stderr

#include <stdio.h>

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

\[ \text{insert 3} \]

\[ \text{insert 1} \]

\[ \text{delete} \]

\[ \text{delete} \]
Implementation 2 of queues

Array

3 1 4 1 5 9 2 3

f

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: $O(1)$ (at front)
   search: $O(n)$
   pseudo-data: target

Rep 2: Sorted linked list
   pseudo-data: $\infty$
   search: $O(n)$

Rep 3: Array
   counter

   insert: $O(1)$
   search: $O(n)$
Rep 4: Sorted array

counter

insert: $O(n)$

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

insert at end:

32 49 56 83 92

70 70 92

70 83

insert at start

look from start

70 83 92
Recursion theorem

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

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

Then

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

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

Root:

- 42

Internal node:

- 45
  - 43
  - 100

Leaf:

- 7
  - 2
  - 1
  - 3
  - 4
  - 5
  - 6
  - 10
  - 77
  - 100
  - 100
  - 803
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>(n \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

   \[ \rightarrow \text{one value at a time} \]

   \[ \rightarrow \text{have all values in advance} \]

   \[ \Rightarrow \text{build a balanced tree} \]

   \[ \text{in } O(n \log n) \]

3. Search (exact, close)

4. Traversals

   \[
   \text{void symmetric(in-order)}
   \]

   \[
   \text{if } \text{node is empty}\}
   \]

   \[
   \text{else}\}
   \]

   \[
   \text{node} \rightarrow \text{left}
   \]

   \[
   \text{symmetric(left-child)};
   \]

   \[
   \text{visit(node)};
   \]

   \[
   \text{symmetric(node} \rightarrow \text{right});
   \]

   \[
   \}
   \]
```c
void preorder(node)
{
    if (node is empty)
    {
    }
    else
    {
        visit(node);
        preorder(node->left);
        preorder(node->right);
    }
    // not empty
}

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

5. not sorted, want 2nd highest.
   \[ \text{largest} = \text{nextLargest} = -\infty \]
   \[
   \text{foreach value in array} \ \\
   \text{if (value > largest)} \ \\
   \text{nextLargest} = \text{largest}; \ \\
   \text{largest} = \text{value};
   \]
   \[
   \text{else if (value > nextLargest)} \ \\
   \text{nextLargest} = \text{value};
   \]
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.

\[
\begin{array}{c}
\text{arbitrary} \\
\uparrow \\
\text{small} | \text{P_1} | \text{large} \\
\uparrow \\
x \quad x \\
\quad \quad \quad \text{after 1 partitioning} \\
\uparrow \\
\text{X} \\
\quad \quad \quad \text{after 2 partitioning} \\
\uparrow \\
\text{P} \\
\uparrow \\
\end{array}
\]

Complexity:

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

recurrence relation

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

\[
b = 2 \quad k = 1
\]
$\Theta(1)$ constant
$\Theta(\log n)$ logarithmic
$O(n)$ linear
$O(n^2)$ quadratic
$O(n^3)$ cubic
$O(2^n)$ exponential

Nico Lomuto's algorithm
```
"comb" method.

pivot_value = array[0]
\[ p = 0 \]
for (\[ p \]...)
```

```plaintext
l : l = index
p : pivot index
```
Sorting methods

Insertion Sort:

\[ \Theta(n^2) \]

\[ \text{before} \]

\[ \text{after} \]
Selection sort

\[ \Theta(n^2) \]

QuickSort (C. A. R. Hoare)

\[ \text{initial} \]

\[ \text{partition} \]

\[ \text{after 1 step} \]

\[ \text{recurse on both sides} \]
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

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

\[ \begin{align*}
\text{partition} & : k = 1 \\
\alpha = 2 \\
\beta = 2 \\
\theta(n^{1 + 2 \frac{n/2}{2}}) & : b^k = 2 \\
\Theta(n^k \log n) = \Theta(n / \log n)
\end{align*} \]
Consider a worse partitioning

\[ C_n = n^1 + 2^n C_n^{1/3} \]

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

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

Shell Sort (Donald Shell 1959)

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

For each offset \( 0 \cdots 2s-1 \)
Next pass: smaller span.

\[
\begin{array}{c}
\begin{array}{c}
\vdots \\
3 \\
\vdots \\
\end{array}
\end{array}
\]

offset 0

offset 1

last pass: \( \text{span} = 1 \)

Heaps

\[
\begin{array}{c}
\begin{array}{c}
42 \\
3 \\
\end{array}
\end{array}
\]

→

2

\[
\begin{array}{c}
\begin{array}{c}
3 \\
42 \\
\end{array}
\end{array}
\]

→

3

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

→

3

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

→

2

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

→

3

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

→

43

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

Fast heapification

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

Know the range of key
Know no duplicates

Array of bits

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

Array of bits

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

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 $m$ digits (decimal), $n$ numbers

| 100 | 200 | 369 | 927 | 150 | 062 | 042 | 746 |

Cost: $j$ passes

| 100 | 062 | 945 | 927 | 369 |
| 200 | 042 |
| 150 |

$O(jn)$

j is no bigger than log n

$O(\log n)$

| 042 | 100 | 746 | 369 | 946 | 062 | 150 | 062 | 150 | 062 | 150 |
\[ C_n = n + \frac{1}{2} C_{n/2} \]
\[ n \leq k \]
\[ b \]
\[ \Theta(n^k \log m) = \Theta(n \log n) \]
\[ \text{guaranteed.} \]
\[ \text{stable} \]

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: \( x \)

\[ \text{right} \]

\[ a \]

\[ \text{left} \]

\[ b \]

\[ c \]
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

case 1: parent and uncle are red.
circle! black  

\[
\begin{array}{c}
\circ \quad 9 \\
p \\
u \\
v
\end{array}
\xrightarrow{\text{reco}}
\begin{array}{c}
P \\
v \\
u
\end{array}
\]

case 2: parent red, uncle black, v inside

\[
\begin{array}{c}
\circ \quad 9 \\
\circ \quad u \\
p \\
v \\
\circ \quad 3 \\
1 \\
2
\end{array}
\xrightarrow{\text{rotate}}
\begin{array}{c}
P \\
u \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{v: up}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{p: down}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{g: to}}
\text{case 3}

case 3: parent red, uncle black v outside

\[
\begin{array}{c}
\circ \quad 9 \\
\circ \quad u \\
p \\
\circ \quad u \\
\circ \quad v \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{reco}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{rotate}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{v: up}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{g: down}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{p: down}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
\xrightarrow{\text{g:}}
\begin{array}{c}
P \\
\circ \quad u \\
\circ \quad 9 \\
1 \\
2 \\
3
\end{array}
Insert 1, 2, 3, 4, 5, 6

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

1. Insert 5
2. Insert 2
3. Insert 7
4. Insert 4
5. Insert 3
6. Insert 1

Case 1:
- 6
- 5
- 4
- 3
- 2
- 1

Case 2:
- 5
- 7
- 4
- 3
- 2
- 1

Case 3:
- 5
- 7
- 4
- 3
- 2
- 1

Case 4:
- 6
- 5
- 4
- 3
- 2
- 1

Case 5:
- 6
- 5
- 4
- 3
- 2
- 1

Case 6:
- 6
- 5
- 4
- 3
- 2
- 1

Case 7:
- 6
- 5
- 4
- 3
- 2
- 1

Case 8:
- 6
- 5
- 4
- 3
- 2
- 1

Case 9:
- 6
- 5
- 4
- 3
- 2
- 1

Case 10:
- 6
- 5
- 4
- 3
- 2
- 1

Case 11:
- 6
- 5
- 4
- 3
- 2
- 1

Case 12:
- 6
- 5
- 4
- 3
- 2
- 1

Case 13:
- 6
- 5
- 4
- 3
- 2
- 1

Case 14:
- 6
- 5
- 4
- 3
- 2
- 1

Case 15:
- 6
- 5
- 4
- 3
- 2
- 1

Case 16:
- 6
- 5
- 4
- 3
- 2
- 1

Case 17:
- 6
- 5
- 4
- 3
- 2
- 1

Case 18:
- 6
- 5
- 4
- 3
- 2
- 1

Case 19:
- 6
- 5
- 4
- 3
- 2
- 1

Case 20:
- 6
- 5
- 4
- 3
- 2
- 1

Case 21:
- 6
- 5
- 4
- 3
- 2
- 1

Case 22:
- 6
- 5
- 4
- 3
- 2
- 1

Case 23:
- 6
- 5
- 4
- 3
- 2
- 1

Case 24:
- 6
- 5
- 4
- 3
- 2
- 1

Case 25:
- 6
- 5
- 4
- 3
- 2
- 1

Case 26:
- 6
- 5
- 4
- 3
- 2
- 1

Case 27:
- 6
- 5
- 4
- 3
- 2
- 1

Case 28:
- 6
- 5
- 4
- 3
- 2
- 1

Case 29:
- 6
- 5
- 4
- 3
- 2
- 1

Case 30:
- 6
- 5
- 4
- 3
- 2
- 1

Case 31:
- 6
- 5
- 4
- 3
- 2
- 1

Case 32:
- 6
- 5
- 4
- 3
- 2
- 1
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

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

\[ s = d \text{'s } R \text{ or } L \text{*} \]

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

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

---

Extensions to binary trees

Ternary tree

\[
\begin{array}{c}
42, 50 \\
7, 8 \\
20, 21
\end{array}
\begin{array}{c}
45, 48 \\
7, 90
\end{array}
\begin{array}{c}
85, 60 \\
75, 77
\end{array}
\]

Cost of searching:

- \( \log_3 n \) levels
- 573 comparisons per level

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

Compare to binary tree

- \( \log_2 n \) levels
- 1 comparison/level

\[
\frac{A}{B} = \frac{1}{1.05}
\]
Quad trees

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

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

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

Check first if sorted
\[ C_n = n + 3 c_{\frac{2}{3}n} \]

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

\[
\begin{array}{cccccccc}
3 & 4 & 5 & 9 & 2 & 6 & 5 & 3 \\
1 & 3 & \cdot & \cdot & \cdot \\
1 & 3 & 4
\end{array}
\]

7) insert online into 2-3 tree

\[
1, 3, 4 \rightarrow \begin{array}{cccccccc}
1, 2, 4, 5, 9, 1, 1, 4, 9
\end{array}
\]

f) 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
**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 \( m-1 \) values.
- 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.

\[ \text{insert()} : \Theta(1) \]
\[ \text{search()} : \Theta(1) \]

Idea: find a value \( k \) by looking in an array at location \( h(k) \) via hash function.

Birthday paradox

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

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

\[ \Rightarrow \text{One must deal with collisions.} \]
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 \( (p+1)h(k) \mod s \)
avoid clustering.
\( h(k) \) must never be 0.
Online:

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

739 546 6788

Telephone: 646-876-2923

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

Insert: into the chain associated with \( h(k) \)

Make sure array is at least as big as the number of keys

Average list length is \( \frac{\# \text{ of Keys}}{\#} \)
Optimizations

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

2) For testing: use $h(c) = 0$
   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 $0 \ldots s-1$
Spreading: similar keys should
hash far apart.

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

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 $h(k)$.

   \[
   \text{even (last bit of } h \text{)} \quad \Rightarrow \quad \text{odd ("" )}
   \]

   \[
   \text{last bits 01} \quad \Rightarrow \quad \text{last bits 11}
   \]

"Trie" $\mathcal{O}(\log m)$

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
Spanning Trees:

Cycle-free subgraph containing all vertices.

Minimal-weight spanning tree:
Total edge weight to be minimized.

Prim's algorithm: add vertices, edges to an initially empty tree.
Start with any vertex.
Do \( v-1 \) times:
Add the closest external vertex.
Data structure for Prim's Algorithm

Heap: each element is

- vertex
- key = distance to growing tree
- closest vertex in that tree

As a vertex is added to tree:
- delete from heap
- for each neighbor that is not in the tree:
  - remove from heap
  - if better
  - reinsert with new value

heap: 

1) replace it with last value
2) sift down
3) insert modified value

Complexity: \( \Theta (n^2 \log n) \)
Kruskal's algorithm:

Start with all vertices, no edges.
- Add \( n-1 \) times:
  - add the lowest-weight missing edge that does not cause a cycle.
Union-find algorithm
for cycle detection

General idea:

As edges are added, keep track of which component each vertex belongs to, by selecting a representative.

In considering adding an edge between \( v_1 \) and \( v_2 \), first see if they are already in the same component.

Operations:

- **Union** \((v_1, v_2)\): Puts \( v_1 \) and \( v_2 \) into the same component.

- **Find** \((v, \_\)\): Reports the representative.
Algorithm for union:

data structure for a vertex:

- name (could be int)
- representative: null if me
- int depth

union(U, V, z):

follow the "representative" chain until you find null for both
U \lor V

set U's representative to be \text{null}.

\begin{align*}
&u(1, 2) \\
&u(2, 4) \\
&u(5, 6) \\
&u(6, 4)
\end{align*}

rule: point shallower tree to the deeper
Optimizing path shortening:

as you find a representative, redirect directly bit.