This Course

• Sorting
  – A necessary tool for managing and navigating a variety of datasets.
  – A great context in which to learn computer science
    • Analyze Complexity
    • Prove Correctness

• Data Structures
  – Explore variety of structures beyond arrays and linked lists.
  – How you store data affects how easily you can manage and navigate it.

• Optimization Problems
  – Learn sophisticated methods that are alternates to exponential brute force approach.

• Graph Problems
  – Applicable to wide range of problems (many seemingly unrelated problems map to a graph problem).
  – Another great context in which to learn computer science.
Functionality of Data Structure for Dynamic Sets

Structure Types
- Array, Linked List, Stack, Queue, Ring Buffer
- Hash Table
- Rooted Tree: Heap, Binary Search, Red-Black

Basic Functionality
- $S$ is an ordered set (i.e. already sorted)
- $k$ is key of an element
- $x$ is a pointer

- Search($S, k$) : find element with key $k$ and return pointer.
- Minimum($S$) / Maximum($S$) : return pointer to min/max
- Successor($S, x$) / Predecessor($S, x$): return pointer to successor/predecessor of element pointed to by $x$.
- Insert($S, x$) : add pointed to element to set
- Delete($S, x$) : remove pointed to element (i.e. already found)
The Structures

• Hash Tables
  – chaining and open addressing

• Binary Search Trees
  – cursory understanding of Red-Black trees

• Know the algorithm and hidden implementation of each.
  – search, insert, and delete
  – best, average, and worst time requirements

Data Structures Are Algorithms
```c
int x;
x = 25;
x = x + 1;
```

**Left-hand Side:**
Get ready to put something in this memory location.

**Reference Memory**
(get the address)

**Right-hand Side:**
Get the date at this memory location.

**De-Reference Memory**
(get the data)

- fill memory location “x” with 25
- take data in memory location “x”, add 1, fill memory location “x” with results
Internally, *arrays* are blocks of nondescript memory.

Arrays are de/referenced with pointers (but you don’t see it).

**Hidden Pointer**

```c
int myArray[5];
myArray[0] = 5;
myArray[1] = 10;
result = myArray[1] + 50;
```

**Transparent Pointer**

```c
int *pmyArray = myArray;
*pmyArray = 5;
*(pmyArray + 1) = 10;
*(++pmyArray) = 10;
result = *pmyArray + 50;
result = pmyArray[0] + 50;
```
Array Versus Linked List

Random VS Sequential Access
Fixed VS Dynamic Size

- Array
- Linked Lists
  - head (maybe tail)
  - next : pointer to successor
  - prev: pointer to predecessor (optional but typically useful)
  - Sentinels can be used.
    “Sentinels rarely reduce asymptotic time but can reduce constant factors.”

Memory Management is Extremely Important for Large Data Sets
• Stack (LIFO): all activity happens on the top of the stack
  – push = insert
  – pop = delete
  – typically only look at top (i.e. no searches or ordering)

• Queue (FIFO): take from front (head), add to the back (tail)
  – enqueue = insert
  – dequeue = delete
  – typically only look at head (i.e. no searches or ordering)

• Ring Buffer (FIFO): Different visualization of a Queue
Memory Management and Effect on “Constants”
  • static : globals
  • heap : runtime (dynamic) variables
  • stack : saves “state” (scoped variables)

Large Data Sets
  • recursive versus iterative algorithms
  • garbage collection
    – avoid leaks and dangling pointers.
Functionality of Data Structure for Dynamic Sets

Structure Types

- Array, Linked List, Stack, Queue, Ring Buffer
- Hash Table
- Rooted Tree: Heap, Binary Search, Red-Black

Basic Functionality

S is an ordered set (i.e. already sorted)

k is key of an element

x is a pointer

- Search( S, k ) : find element with key k and return pointer.
- Minimum( S ) / Maximum( S ) : return pointer to min/max
- Successor( S, x ) / Predecessor( S, x ) : return pointer to successor/predecessor of element pointed to by x.
- Insert( S, x ) : add pointed to element to set
- Delete( S, x ) : remove pointed to element (i.e. already found)
Ordered Sets

- Linked Lists
- Arrays
- **Hash Tables**
- Rooted Trees
  - various branching factors (binary common)
  - heap (array implementation of balanced binary tree)
  - binary search trees, red-black trees (balanced binary trees)

**Functions**
- `Search(S, k)`: find element with key k and return pointer.
- `Insert(S, x)`: add pointed to element to set
- `Delete(S, x)`: remove pointed to element (i.e. already found)
- `Minimum(S)` / `Maximum(S)`: return pointer to min/max
- `Successor(S, x)` / `Predecessor(S, x)`: return pointer to successor/predecessor of element pointed to by x.
Direct Addressing and Hashing Tables

**DIRECT ADDRESSING**

```
DIRECT-ADDRESS-SEARCH(T, k)
1 return T[k]
DIRECT-ADDRESS-INSERT(T, x)
1 T[x.key] = x
DIRECT-ADDRESS-DELETE(T, x)
1 T[x.key] = NIL
```

All operations are O(1).
What are the issues with this approach?

**INDIRECT ADDRESSING USING A HASHING FUNCTION**

```
HASH-SEARCH( T, k )
return T[ h(k) ]
HASH-INSERT( T, x )
T[ h( x.key ) ] = k;
HASH-DELETE( T, x )
T[ h( x.key ) ] = NIL;
```

Hashing Function
h: {key range} -> { 0 .. m }
h( 42987 ) = 5;

All operations are O(1).
What are the issues with this approach?
Hashing with Chaining

CHAINED-HASH-INSERT(T, x)
1 insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1 search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1 delete x from the list T[h(x.key)]

Chaining (i.e. linked list) to manage Collisions (i.e. \( h(k_i) = h(k_j) \))

BUCKET-SORT(A)
1 \( n = A.length \)
2 let \( B[0..n-1] \) be a new array
3 for \( i = 0 \) to \( n - 1 \)
4 make \( B[i] \) an empty list
5 for \( i = 1 \) to \( n \)
6 insert \( A[i] \) into list \( B[i] \)
7 for \( i = 0 \) to \( n - 1 \)
8 sort list \( B[i] \) with insertion sort
9 concatenate the lists \( B[0], B[1], \ldots, B[n-1] \) together in
“Unfortunately, we typically have no way to check (that each key is equally likely to hash to any of the m slots), since we rarely know the probability distribution...”

- k numbers uniformly distributed in [0,1)
  \[ h(k) = \text{floor}(km) \text{ is good.} \]
- **division method**: \( h(k) = k \mod m \)
  choosing a prime not too close to exact power of 2 is good.
- **multiplication method**: \( h(k) = \text{floor}( m (kA \mod 1) ) \). \( A = (0,1) \)
  \( A = 0.61803... \) is good
Hashing with Chaining

Hashing Function

CHAINED-HASH-INSERT(T, x)
1 insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1 search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1 delete x from the list T[h(x.key)]

Expected Running Time ?
load factor $\alpha = \frac{m}{n}$
m: # slots
n: # elements

ASSUMING
simple uniform hashing
$E[ n_j ] = \alpha$

ASSUMING
simple uniform hashing
Hashing with Open Addressing
(Alternative to Chaining)

• Open Addressing is like a Queue + Ring Buffer
  – hash into the table
  – if occupied, probe until you find empty slot (wrapping around to front)

• Probe Sequence: \(< h(k,0), h(k,1), h(k,2), ..., h(k,m-1) >\)
  – for every \(k\), this sequence must be a permutation of \(< 0 .. m-1 >\)

• “Deletion from an open-address hash table is difficult”

```
INSERT( T, k )
Repeat
  h = k mod m
  if T(h)== NIL
    T(h) = k; return;
  else
    h = (h + 1) mod m

SEARCH( T, k )
Repeat
  h = k mod m
  if T(h) == NIL return NIL;
  if T(h) == k return k;
  else
    h = (h + 1) mod m
```

Try this:
m=10;
insert 42, 65, 23, 72.
delete 23.
search for 72.
• Linear Probing: \( h(k,i) = (h'(k) + i) \mod m \)
  - hash in to the table using function \( h' \)
  - look in the next slot until you find an open one (mod m)
  - problem with primary clustering

• Quadratic Probing: \( h(k,i) = (h'(k) + c_1i + c_2i^2) \mod m \)
  - hash in to the table using function \( h' \)
  - “jump” to another location based on quadratic equation
  - problem with secondary clustering

• Double Hashing: \( h(k,i) = (h_1(k) + ih_2(k)) \mod m \)
  - one of the best methods
  - keys with equivalent \( h_1(k) \) will not generate the same probe sequence

**uniform hashing**
the probe sequence of each key is equally likely to be any of the \( m! \) permutations.
### Expected Running Time

\[
\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\
\leq \left(\frac{n}{m}\right)^{i-1} \\
= \alpha^{i-1}.
\]

Now, we use equation (C.25) to bound the expected number of probes:

\[
E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\
\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\
= \sum_{i=0}^{\infty} \alpha^{i} \\
= \frac{1}{1-\alpha}.
\]

**Expected Number of Probes in unsuccessful search**

\[E[X] = \frac{1}{1-\alpha}\]

Choose \(m\), so that \(\alpha \leq 0.9\) then expect at most 10 probes at most

\[\text{Search} = O(1)\]

When will search be \(O(n)\), which is the worst case scenario?
Ordered Sets

- Linked Lists
- Arrays
- Hash Tables
- **Rooted Trees**
  - various branching factors (binary common)
  - heap (array implementation of balanced binary tree)
  - binary search trees, red-black trees (balanced binary trees)

**Functions**
- Search( S, k ) : find element with key k and return pointer.
- Insert( S, x ) : add pointed to element to set
- Delete( S, x ) : remove pointed to element (i.e. already found)
- Minimum( S ) / Maximum( S ) : return pointer to min/max
- Successor( S, x ) / Predecessor( S, x ) : return pointer to successor/predecessor of element pointed to by x.
Binary Search Trees

Binary Search Property

x is a node

if y in x.left subtree, y.key ≤ x.key

if y in x.right subtree, y.key ≥ x.key

INORDER-TREE-WALK(x)
1 if x ≠ NIL
2 INORDER-TREE-WALK(x.left)
3 print x.key
4 INORDER-TREE-WALK(x.right)
Search

TREE-SEARCH(x, k)
1 if x == NIL or k == x.key
2 return x
3 if k < x.key
4 return TREE-SEARCH(x.left, k)
5 else return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k)
1 while x ≠ NIL and k ≠ x.key
2 if k < x.key
3 x = x.left
4 else x = x.right
5 return x

TREE-MINIMUM(x)
1 while x.left ≠ NIL
2 x = x.left
3 return x

TREE-MAXIMUM(x)
1 while x.right ≠ NIL
2 x = x.right
3 return x

TREE-SUCCESSOR(x)
1 if x.right ≠ NIL
2 return TREE-MINIMUM(x.right)
3 y = x.p
4 while y ≠ NIL and x == y.right
5 x = y
6 y = y.p
7 return y
**Insert and Delete**

```
TREE-INSERT(T, z)
1 y = NIL
2 x = T.root
3 while x ≠ NIL
4 y = x
5 if z.key < x.key
6 x = x.left
7 else x = x.right
8 z.p = y
9 if y == NIL
10 T.root = z  // tree T was empty
11 elseif z.key < y.key
12 y.left = z
13 elseif y.right = z
```

“Transplant replaces the subtree rooted at node u with the subtree nrooted at node v.”

```
TRANSPLANT(T, u, v)
1 if u.p == NIL
2 T.root = v
3 elseif u == u.p.left
4 u.p.left = v
5 else u.p.right = v
6 if v ≠ NIL
7 v.p = u.p
```

```
TREE-DELETE(T, z)
1 if z.left == NIL
2 TRANSPLANT(T, z, z.right)
3 elseif z.right == NIL
4 TRANSPLANT(T, z, z.left)
5 elseif y = TREE-MINIMUM(z.right)
6 if y.p ≠ z
7 TRANSPLANT(T, y, y.right)
8 y.right = z.right
9 y.right.p = y
10 TRANSPLANT(T, z, y)
11 y.left = z.left
12 y.left.p = y
```
Expected Height

O(h)!
“Unfortunately, little is known about the average height of a binary search tree when both insertion and deletion are used to create it.

Theorem 12.4
The expected height of a randomly built binary search tree on n distinct keys is O(lg n).

\[ h = X_n \]

\[ Y_n = 2^X_n \]

\[ Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}). \]

\[ h = 1 + \text{max height of the 2 subtrees of the root} \]

\[ i = R_n = \text{rank of root} \]

\[ Z_{ni} = I\{ R_n = i \} \]

\[
E[Y_n] = E \left[ \sum_{i=1}^{n} Z_{n,i} \left( 2 \cdot \max(Y_{i-1}, Y_{n-i}) \right) \right]
\]
Red-Black Trees

- Binary Search Trees with color
- Uses color scheme to maintain relative balance
- Extra work to insert and delete to maintain balance.

\[
\text{Best( Binary )} < \text{Best( Red-Black )} \{ \text{by constant factor} \}
\]

\[
\text{Worst( Binary )} > \text{Worst( Red-Black )} \{ O(n) \text{ versus } O(\log n) \}
\]

\[
\text{Average( Binary )} = \text{Worst( Red-Black )}
\]

Guaranteed Bound on the Height
Red-Black Trees ("Balanced" Binary Tree)

- All nodes on left are ≤, all on right are ≥ (i.e. binary search tree).
- Every node is either red or black.
- Root is black.
- Every leaf (NIL) is black.
- If node red, both its children are black.
- For each node, all simple paths from node to leaf contain same number of black nodes.

Height at most $2\times \lg(n+1)$
Key to Balance: Rotation

\[ \alpha \leq x \leq \beta \leq y \leq y \]

LEFT-ROTATE\((T, x)\)

1. \(y = x.\text{right}\)
2. \(x.\text{right} = y.\text{left}\)
3. if \(y.\text{left} \neq T.\text{nil}\)
4. \(y.\text{left}.p = x\)
5. \(y.p = x.p\)
6. if \(x.p == T.\text{nil}\)
7. \(T.\text{root} = y\)
8. elseif \(x == x.p.\text{left}\)
9. \(x.p.\text{left} = y\)
10. else \(x.p.\text{right} = y\)
11. \(y.\text{left} = x\)
12. \(x.p = y\)

- Reduced height by 1
- Nodes distributed more evenly
\[ \alpha \leq x \leq \beta \leq y \leq \gamma \]

**Insert:** \(O(lg n)\)

**Delete:** \(O(lg n)\)

Height at most \(2*lg(n+1)\)

---

**RB-INSERT** \((T, z)\)

1. \(y = T.nil\)
2. \(x = T.root\)
3. **while** \(x \neq T.nil\)
   4. \(y = x\)
   5. **if** \(z.key < x.key\)
      6. \(x = x.left\)
   7. **else** \(x = x.right\)
   8. \(z.p = y\)
   9. **if** \(y == T.nil\)
      10. \(T(root) = z\)
   11. **elseif** \(z.key < y.key\)
      12. \(y.left = z\)
   13. **else** \(y.right = z\)
   14. \(z.left = T.nil\)
   15. \(z.right = T.nil\)
   16. **if** \(z.color == RED\)
      17. \(z.color = BLACK\)
      18. \(LEFT-ROTATE(T, z)\)
      19. \(z.p.color = BLACK\)
      20. \(z.p.p.color = RED\)
      21. \(RIGHT-ROTATE(T, z.p.p)\)
      22. \(T.root.color = BLACK\)

---

**RB-INSERT-FIXUP** \((T, z)\)

1. **while** \(z.p.color == RED\)
   2. **if** \(z.p == z.p.p.left\)
      3. \(y = z.p.p.right\)
      4. **if** \(y.color == RED\)
         5. \(z.p.color = BLACK\)
         6. \(y.color = BLACK\)
         7. \(z.p.p.color = RED\)
         8. \(z = z.p.p\)
   9. **else if** \(z == z.p.right\)
      10. \(z = z.p\)
      11. \(LEFT-ROTATE(T, z)\)
      12. \(z.p.color = BLACK\)
      13. \(z.p.p.color = RED\)
      14. \(RIGHT-ROTATE(T, z.p.p)\)
      15. **else** (same as then clause with "right" and "left" exchanged)

---

- If node red, both children black.
- For each node, path to each leaf contains same # black nodes.
• Hash Tables
  – chaining and open addressing

• Binary Search Trees
  – cursory understanding of Red-Black trees

• Know the algorithm and hidden implementation of each
  – search, insert, and delete
  – best, average, and worst time requirements

• Memory Management
  – static, heap, stack – effect on time and space.
  – recursive versus iterative algorithms.
  – large data sets require attention to memory management

Data Structures Are Algorithms