Culmination of the programming portion of this class.

- Solve a database search problem.

Tree data structure.
- Useful.
- Versatile.
- Naturally recursive.

Searching a Database

Database entries.
- Names and social security numbers.

Desired operations.
- Insert student.
- Delete student.
- Search for name given ID number.

Goal.
- All operations fast, even for huge databases.

Data structure that supports these operations is called a SYMBOL TABLE.

<table>
<thead>
<tr>
<th>SS #</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>192042006</td>
<td>Arac</td>
</tr>
<tr>
<td>201211991</td>
<td>Baron</td>
</tr>
<tr>
<td>177998898</td>
<td>Bergbreiter</td>
</tr>
<tr>
<td>232871212</td>
<td>Buchen</td>
</tr>
<tr>
<td>122993434</td>
<td>Durrett</td>
</tr>
<tr>
<td>162882273</td>
<td>Gratzer</td>
</tr>
</tbody>
</table>

“search key”

Other Symbol Table Applications

Other applications.
- Online phone book looks up names and telephone numbers.
- Spell checker looks up words in dictionary.
- Internet domain server looks up IP addresses.
- Compiler looks up variable names to find type and memory address.

<table>
<thead>
<tr>
<th>Web Site</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.cs.princeton.edu">www.cs.princeton.edu</a></td>
<td>128.112.136.11</td>
</tr>
<tr>
<td><a href="http://www.princeton.edu">www.princeton.edu</a></td>
<td>128.112.128.15</td>
</tr>
<tr>
<td><a href="http://www.yale.edu">www.yale.edu</a></td>
<td>130.132.143.21</td>
</tr>
<tr>
<td><a href="http://www.harvard.edu">www.harvard.edu</a></td>
<td>128.103.060.55</td>
</tr>
<tr>
<td><a href="http://www.amazon.com">www.amazon.com</a></td>
<td>208.216.181.15</td>
</tr>
<tr>
<td><a href="http://www.pregnantchad.com">www.pregnantchad.com</a></td>
<td>209.052.165.60</td>
</tr>
</tbody>
</table>
Representing the Database Entries

Define Item.h file to encapsulate generic database entry.
- Insert and search code should work for any item type.
  - Ideally Item would be an ADT
- Key is field in search.

```c
#define MAXSIZE 10000
static Item st[MAXSIZE];
static int size = 0;

Item STinsert(Item item) {
  st[size] = item;
  size++;
}

Item STsearch(Key k) {
  int i;
  for (i = 0; i < size; i++)
    if eq(k, key(st[i]))
      return st[i];
  return NULLitem;
}
```

Symbol Table ADT

Define ST.h file to specify database operations.
- Make it a true symbol table ADT.

```c
#include "ITEM.h"
int eq(Key k1, Key k2) {
  return k1 == k2;
}
int less(Key k1, Key k2) {
  return k1 < k2;
}
Key key(Item x) {
  return x.ID;
}
void show(Item x) {
  printf("%d %s\n", x.ID, x.name);
}
```

Unsorted Array Representation of Database

Maintain array of Items.
- Use SEQUENTIAL SEARCH to find database Item.

Advantage.
- Insertion is fast.
- Need to fix maximum database size ahead of time.

Extra problem.
- Need to look at every database entry if Key not found.

```
#define MAXSIZE 10000
static Item st[MAXSIZE];
static int size = 0;

Item STinsert(Item item) {
  st[size] = item;
  size++;
}

Item STsearch(Key k) {
  int i;
  for (i = 0; i < size; i++)
    if eq(k, key(st[i]))
      return st[i];
  return NULLitem;
}
```
**Sorted Array Representation of Database**

Maintain array of Items.
- Store in sorted order (by Key).
- Use BINARY SEARCH to find database Item.

```c
#define MAXSIZE 10000
static Item st[MAXSIZE];
static int size = 0;

Item search(int l, int r, Key k) {
    int m = (l + r) / 2;
    if (l > r)
        return NULLitem;
    else if eq(k, key(st[m]))
        return st[m];
    else if less(k, key(st[m]))
        return search(l, m-1, k);
    else
        return search(m+1, r, k);
}
```

**STsortedarray.c (Sedgewick 12.6)**

"Wrapper" for search function.

```c
Item STsearch(Key k) {
    return search(0, size-1, k);
}
```

**Cost of Binary Search**

How many "comparisons" to find a name in database of size N?
- Divide list in half each time.
  - 5000 2500 1250 625 312 156 78 39 18 9 4 2 1
  - $\lceil \log_2 (N+1) \rceil$ = number of digits in binary representation of N.
  - 5000 = 1001100001000_2

The log functions grows very slowly.
- $\log_2$ (thousand) ≈ 10
- $\log_2$ (million) ≈ 20
- $\log_2$ (billion) = 30

Without binary search (or if unsorted): may examine all N items.
- N vs. $\log_2$ N savings is staggering for large files.
- Milliseconds vs. hours (or more!).

**Insert Using Sorted Array Representation**

Key Problem: insertion is slow.
- Want to keep entries in sorted order.
- Have to move larger keys over one position to right.

Demo: inserting 25 into a sorted array.
Sorted Array Representation of Database

Maintain array of Items.
- Store in sorted order (by key).
- Use BINARY SEARCH to find database item.

Advantage.

Key drawback.

Extra problem.

Summary

Database entries.
- Names and social security numbers.

Desired operations.
- Insert, delete, search.

Goal.
- Make all of these operations FAST even for huge databases.

<table>
<thead>
<tr>
<th>asymptotic time</th>
<th>computer time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>search</strong></td>
<td><strong>insert</strong></td>
</tr>
<tr>
<td>sorted array</td>
<td>log N</td>
</tr>
<tr>
<td>unsorted array</td>
<td>N</td>
</tr>
<tr>
<td><strong>goal</strong></td>
<td>log N</td>
</tr>
</tbody>
</table>

Is there any way to have fast insert AND search?

Binary Tree

Yes. Use TWO links per node!

Root: 14
- Parent: 43
  - Left child: 33
    - Left child: 13
    - Right child: 64
      - Left child: 53
      - Right child: 99
        - Left child: 7
        - Right child: 64
          - Right child: 51
          - Left child: 25

Binary Tree in C

STbst.h

typedef struct STnode* link;
struct STnode {
  Item item;
  link left;
  link right;
};

static link root;

Represent in C with TWO links per node.
- Leftmost arrow corresponds to left link.
- Rightmost to right link.
Binary Search Tree

- Binary tree in "sorted" order.
  - Maintain ordering property for ALL sub-trees.

left subtree (smaller values)  
right subtree (smaller values)

root (middle value)

Binary tree in "sorted" order.
- Maintain ordering property for ALL sub-trees.

Many BST's for the same input data.
- Have different tree shapes.

Binary tree in "sorted" order.
- Maintain ordering property for ALL sub-trees.
Search in Binary Search Tree

Search for key \( k \) in binary search tree.
- Analogous to binary search in sorted array.

Search algorithm:
- Start at head node.
- If key of current node is \( k \), return node.
- Go LEFT if current node has Key < \( k \).
- Go RIGHT if current node has Key > \( k \).

Cost of BST Search

Depends on tree shape.
- Proportional to length of path from root to Key.
- "Balanced."
  - \( 2 \log_2 N \) comparisons
  - proportional to binary search cost

- "Unbalanced."
  - takes \( N \) comparisons for degenerate tree shapes
  - can be as slow as sequential search

Algorithm works for any tree shape.
- With cleverness (e.g., "red-black trees" in COS 226),
  can ensure tree is always (roughly) balanced.

Insert Using BST's

How to insert new database Item.
- Search for key of database Item.
- Search ends at NULL pointer.
- New Item "belongs" here.
- Allocate memory for new Item, and link it to tree.
Insert Using BST’s

BST.c (Sedgewick 12.7)

// Insert new node here.

link insert(link x, Item item) {
    if (x == NULL)
        return NEWnode(item, NULL, NULL);
    else if (less(key(item), key(x->item))
        x->left = insert(x->left, item);
    else
        x->right = insert(x->right, item);
    return x;
}

void STinsert(Item item) {
    head = insert(root, item);
}

// Divide-and-conquer.

// Wrapper function.

Insertion Cost in BST

- Depends on tree shape.
  - Cost is proportional to length of path from root to node.

- Tree shape depends on order keys are inserted.
  - Insert in “random” order.
    - leads to “well-balanced” tree
    - average length of path from root to node is 1.44 \( \log_2 N \)
  - Insert in sorted or reverse-sorted order.
    - degenerates into linked list
    - takes \( N - 1 \) comparisons

Algorithm works for any tree shape.
- With cleverness (e.g., red-black trees in COS 226),
  can ensure tree is always balanced.

Insert Using BST’s

BST.c (Sedgewick 12.7)

// Allocate memory and initialize.

link NEWnode(Item item, link left, link right) {
    link x = malloc(sizeof *x);
    if(x == NULL) {
        printf("Error allocating memory.\n");
        exit(EXIT_FAILURE);
    }
    x->item = item;
    x->left = left;
    x->right = right;
    return x;
}

// Wrapper function.

Summary

- Database entries.
  - Names and social security numbers.

- Desired operations.
  - Insert, delete, search.

- Goal.
  - Make all of these operations FAST even for huge databases.

<table>
<thead>
<tr>
<th></th>
<th>asymptotic time</th>
<th>computer time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted array</td>
<td>search ( \log N )</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>insert ( N )</td>
<td>2 hour</td>
</tr>
<tr>
<td></td>
<td>delete ( N )</td>
<td>2 hour</td>
</tr>
<tr>
<td>unsorted array</td>
<td>search instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>insert 2 hour</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>delete 2 hour</td>
<td>instant</td>
</tr>
<tr>
<td>BST</td>
<td>search ( \log N )</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>insert ( \log N )</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>delete ( \log N )</td>
<td>instant</td>
</tr>
</tbody>
</table>
Question

Current code searches for a name given an ID number.

What if we want to search for an ID number given a name?

```c
#include <string.h>

int eq(Key k1, Key k2) {
    return strcmp(k1, k2) == 0;
}

int less(Key k1, Key k2) {
    return strcmp(k1, k2) < 0;
}

Key key(Item item) {
    return item.name;
}
```

```c
typedef char Key[30];
typedef struct {
    int ID;
    Key name;
    } Item;

Item NULLItem = {-1, ""};
int eq(Key, Key);
int less(Key, Key);
Key key(Item);
```

Other Types of Trees

Trees.
- Nodes need not have exactly two children.
- Order of children may not be important.

Examples.
- Family tree.
- Parse tree.
- Unix file hierarchy.
  - not binary

```
(a * (b + c)) - (d + e)
```

Charles  mom
Philip
Andrew
Alice
George VI
Elizabeth I
George I
Olga
Louis
Victoria
George V
Mary
Claude
Celia

bin
lib
etc
u

aaclarke
cs126
znye
files
grades
submit
mandel
stock
tsp

POINT.h
point.c
tsp13509.txt

Other Types of Trees

Trees.
- Nodes need not have exactly two children.
- Order of children may not be important.

Examples.
- Family tree.
- Parse tree.
- Unix file hierarchy.
  - not binary

```
(a * (b + c)) - (d + e)
```
Traversing Binary Trees

Goal: visit (process) each node in tree in some order.
- "Tree traversal."
  - Goal realized no matter what order nodes are visited.
    - inorder: visit between recursive calls
    - preorder: visit before recursive calls
    - postorder: visit after recursive calls
Traversing Binary Trees

Goal: visit (process) each node in tree in some order.
- "Tree traversal."
- Goal realized no matter what order nodes are visited.
  - inorder: visit between recursive calls
  - preorder: visit before recursive calls
  - postorder: visit after recursive calls

Preorder Traversal With Explicit Stack

Visit the top node on the stack.
- Push its children onto stack.

```c
void traverse(link x) {
    STACKpush(x);
    while (!STACKempty()) {
        h = STACKpop();
        show(x->item);
        if (x->right != NULL)
            STACKpush(x->right);
        if (x->left != NULL)
            STACKpush(x->left);
    }
}
```

Level Traversal With Queue

Q. What happens if we replace stack with QUEUE?
- Level order traversal.
- Visit nodes in order from distance to root.

```c
void traverse(link x) {
    QUEUEput(x);
    while (!QUEUEisempty()) {
        x = QUEUEget();
        show(x->item);
        if (x->left != NULL)
            QUEUEput(x->left);
        if(x->right != NULL)
            QUEUEput(x->right);
    }
}
```

Summary

How to insert and search a database using:
- Unsorted array.
- Sorted array.
- Binary search tree.

Performance characteristics using different data structures.

Preorder, inorder, postorder, levelorder tree traversals.