Overview

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>177999889</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.

```
# include "ITEM.h"

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

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

Key key(Item item) {
    return item.ID;
}

void show(Item item) {
    printf("%d %s\n", item.ID,
            item.name);
}
```

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

Item NULLItem = {-1, ""};
```

Symbol Table ADT

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

```
ST.h (Sedgewick 12.1)

Item STsearch(Key k); // search for Key in database
void STinsert(Item item); // insert new Item into database
void STshow(void); // print all Items in database
int STcount(void); // number items in database
void STdelete(Item item); // delete Item from database
```

Unsorted Array Representation of Database

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

```
#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;
}
```

Advantage.

Key drawback.

Extra problem.
Sorted Array Representation of Database

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

```
#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);
}
```

Array of database Items.

STsortedarray.c (Sedgewick 12.6)

Key k not found.

Key k found.

Divide-and-conquer.

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
- \(\lfloor \log_2 (N+1) \rfloor\) = number of digits in binary representation of N.
- \(5000_{10} = 100110001000_2\)

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.

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

"Wrapper" for search function.

STsortedarray.c (Sedgewick 12.6)

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>search</td>
<td>insert</td>
</tr>
<tr>
<td>sorted array</td>
<td>log N</td>
</tr>
<tr>
<td>unsorted array</td>
<td>N</td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
</tr>
</tbody>
</table>

* assumes we know location of node to be deleted

Binary Tree

Yes. Use TWO links per node!

Binary Tree in C

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.

root (middle value)

left subtree (smaller values)

right subtree (smaller values)

Binary Search Tree

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

Many BST's for the same input data.
- Many BST's for the same input data.
- Have different tree shapes.
Search in Binary Search Tree

Search for $k$ in binary search tree.
- Analogous to binary search in sorted array.

Search algorithm:
- Start at head node.
- If $k$ 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

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.

Summary

Database entries.
- Names and social security numbers.

Desired operations.
- Insert, delete, search.

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

Asymptotic time

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted array</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
</tbody>
</table>

Computer time

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted array</td>
<td>instant</td>
<td>2 hour</td>
<td>2 hour</td>
</tr>
<tr>
<td>unsorted array</td>
<td>$N$</td>
<td>1</td>
<td>1 $^*$</td>
</tr>
<tr>
<td>BST</td>
<td>2 hour</td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

$^*$ assumes we know location of node to be deleted
Question

Current code searches for a name given an ID number.

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

```
#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;
}
```

```
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);
```
Traversing Binary Trees

Goal: visit (process) each node in tree in some order.
- "Tree traversal."

```
void traverse(link x) {
    if (x == NULL)
        return;
    traverse(x->left); show(x->item); 
    traverse(x->right);
}
```

STbst.c

wrapper function

void STprint(void) {
    traverse(root);
}

inorder

```
void traverse(link x) {
    if (x == NULL)
        return;
    traverse(x->left);
    show(x->item);
    traverse(x->right);
}
```

inorder

```
void traverse(link x) {
    if (x == NULL)
        return;
    traverse(x->left);
    show(x->item);
    traverse(x->right);
}
```

preorder

```
void traverse(link x) {
    if (x == NULL)
        return;
    show(x->item);
    traverse(x->left);
    traverse(x->right);
}
```

postorder

```
void traverse(link x) {
    if (x == NULL)
        return;
    traverse(x->left);
    traverse(x->right);
    show(x->item);
}
```
Traversing Binary Trees

Goal: visit (process) each node in the 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.

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

Preorder traversal with stack

Push right node before left, so that left node is visited first.

Level Traversal With Queue

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

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);
  }
}

Level traversal with queue

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.