**Overview**

Culmination of the programming portion of this class.
- Solve a database search problem.

**Tree data structure.**
- Versatile and useful.
- Naturally recursive.
- Application of stacks and queues.

---

**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>1920342006</td>
<td>Arac</td>
</tr>
<tr>
<td>2012121991</td>
<td>Baron</td>
</tr>
<tr>
<td>1779999898</td>
<td>Bergbreiter</td>
</tr>
<tr>
<td>2328761212</td>
<td>Buchen</td>
</tr>
<tr>
<td>1229993434</td>
<td>Durrett</td>
</tr>
<tr>
<td>1628822273</td>
<td>Gratzer</td>
</tr>
</tbody>
</table>

"search key"
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 x) {
    return x.ID;
}

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

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

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

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

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

Item STsearch(Key k) {
    int i;
    for (i = 0; i < N; 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.

```
ST.h (Sedgewick 12.1)
Item STsearch(Key); /* search for Key in database */
void STinsert(Item); /* insert new Item into database */
void STshow(void); /* print all Items in database */
int STcount(void); /* number items in database */
void STdelete(Item); /* delete Item from database */
```

Unsorted Array Representation of Database

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

```
STunsortedarray.c
#define MAXSIZE 10000
Item st[MAXSIZE];
int N = 0;
```

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

```
Item STsearch(Key k) {
    int i;
    for (i = 0; i < N; 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.

```
#define MAXSIZE 10000
Item st[MAXSIZE];

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)

Array of database Items.

Key k not found.
Key k found.
Divide-and-conquer.

STsortedarray.c (Sedgewick 12.6)

"Wrapper" for search function.

```
Item STsearch(Key k) {
    int N = Stcount();
    return search(0, N-1, k);
}
```

STsortedarray.c  (Sedgewick 12.6)

Array of database Items.

Key k not found.
Key k found.
Divide-and-conquer.

```
#define MAXSIZE 10000
Item st[MAXSIZE];

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.

```
Item STsearch(Key k) {
    int N = Stcount();
    return search(0, N-1, k);
}
```

Sorted Array Representation of Database

Advantage: searching is fast.

Key drawback: inserting is slow.

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 ⇒ 19 ⇒ 9 ⇒ 4 ⇒ 2 ⇒ 1
- $|\log_2 N| = \text{number of digits in binary representation of N.}$
- $5000_{10} = 1001110001000_2$

The log functions grows very slowly.
- $\log_2 (\text{thousand}) \approx 10$
- $\log_2 (\text{million}) \approx 20$
- $\log_2 (\text{billion}) \approx 30$

Without binary search (or if unsorted): may need to look at all N items.
- N vs. $\log_2 N$ savings is staggering for large files.
- Milliseconds vs. years.
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.

| 4 | 6 | 14 | 20 | 25 | 26 | 32 | 47 | 55 | 56 | 58 | 82 |

Demo: inserting 25 into a sorted array.

Problem 2: need to fix maximum database size ahead of time.

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>

Binary Tree

Yes. Use TWO links per node.

Root: 14
Parent: 43
Left child: 64
Right child: 97
Leaf: 64
Binary Tree in C

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

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.

```
  51
 / \   /
 14 21 32
 / \ /   
 66 19 32
 /   /   
 NULL NULL NULL
```

root (middle value)
left subtree (smaller values)
right subtree (smaller values)
Binary Search Tree

Binary tree in "sorted" order.
- Many BST's for the same input data.
- Have different tree shapes.

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.

Search in BST’s

Search for Key k.

```
STbst.c (Sedgewick 12.7)
Item search (link h, Key k) {
  if (h == NULL)
    return NULLitem;
  else if (eq(k, key(h->item))
    return h->item;
  else if (less(k, key(h->item))
    return search(h->left, k);
  else
    return search(h->right, k);
}
```

```
Item STsearch(Key k) {
  return search(head, 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 (see COS 226), can ensure tree is always 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.

```
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;
}
```

BST.c (Sedgewick 12.7)

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

Insert Using BST’s

```
link insert(link h, Item item) {
    Key k  = key(item);
    Key k2 = key(h->item);
    if (h == NULL)
        return NEWnode(item, NULL, NULL);
    else if (less(k, k2))
        h->left = insert(h->left, item);
    else
        h->right = insert(h->right, item);
    return h;
}
```

```
BST.c (Sedgewick 12.7)
```

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 log₂ N
- Insert in sorted or reverse-sorted order.
  - degenerates into linked list
  - takes N - 1 comparisons

Algorithm works for any tree shape.
- With cleverness (see COS 226), can ensure tree is always balanced.
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
// ITEM.h
typedef char Key[30];
typedef struct {
    int ID;
    Key name;
} Item;
Item NULLItem = {-1, ""};

// item.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;
}
```

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

```
Charles   mom
  
    dad

Philip   Elizabeth II
  
    Andrew   Alice
          
      George VI   Elizabeth I
        
    George I   Olga   Louis   Victoria   George V   Mary   Claude   Celia
```

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

```
/   bin   lib   etc   u
  
/  aaclarke   cs126   zmye
       
/  files   grades   submit
       
/  mandel   stock   tsp
       
/  POINT.h   point.c   tsp13509.txt
```
Traversing Binary Trees

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

STbst.c

```c
void STprint(void) {
    traverse(head);
}
```

wrappper function

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

inorder

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

traverse left subtree

traverse right subtree

process node h

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

preorder

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

inorder: visit between recursive calls

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

postorder

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

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

Important note: inorder traversal of BST gives free sort!

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 h) {
    QUEUEput(h);
    while (!QUEUEisempty()) {
        h = QUEUEget();
        show(h->item);
        if (h->left != NULL)
            QUEUEput(h->left);
        if (h->right != NULL)
            QUEUEput(h->right);
    }
}
```

Preorder traversal with stack
Push right node before left, so that left node is visited first.

Summary

How to insert and search a database using:
- Arrays.
- Linked lists.
- Binary search trees.

Performance characteristics using different data structures.
The meaning of different traversal orders and how the code for them works.
Linked List Representation of Database

Keep items in a linked list.
- Store in sorted order.

Insert.
- Only need to change links.
- No need to "move" large amounts of data.

Search.
- Can't use binary search since no DIRECT access to middle element.
- Use sequential search.
  - may need to search entire linked list to find desired Key
  - much slower than binary search

```
typedef struct node* link;
struct node {
    Item item;
    link next;
}
```

```
Item STsearch(Key k) {
    link x;
    for (x = head; x != NULL; x = x->next)
        if (eq(k, key(x))
            return x->item;
    return NULLItem;
}
```