Lecture P11: Trees

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.

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.simpsons.com">www.simpsons.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
#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);
}
```

**ITEM.h**

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

Item NULLItem = {-1, ""};

```c
int eq(Key k1, Key k2);  
int less(Key k1, Key k2);  
Key key(Item item);  
void show(Item item);
```

Symbol Table ADT

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

```c
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`.

```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;
}
```

**STunsortedarray.c**

```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;
}
```

Unsorted Array Representation of Database

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

- 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.

Search algorithm:
- Examine the middle Key.
- If it matches, then we’re done.
- Otherwise, search either the left or right half.
- (A loop would work fine, but this begs for recursion!)

STsortedarray.c (Sedgewick 12.6)

```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);
}
```

Cost of Binary Search

How many “comparisons” to find a name in database of size N?
- Divide list in half each time.
  - $5000 \Rightarrow 2500 \Rightarrow 1250 \Rightarrow 625 \Rightarrow 312 \Rightarrow 156 \Rightarrow 78 \Rightarrow 39 \Rightarrow 18 \Rightarrow 9 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$
- $\lceil \log_2 (N+1) \rceil = \text{number of digits in binary representation of } N.$
  - $5000_{10} = 1001110001000_2$

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

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!).

"Wrapper" for search function.

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

---

Binary Tree

**Yes. Use TWO links per node!**

---

<table>
<thead>
<tr>
<th></th>
<th>asymptotic time</th>
<th>computer time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
</tr>
<tr>
<td><strong>sorted array</strong></td>
<td>log N</td>
<td>N</td>
</tr>
<tr>
<td><strong>unsorted array</strong></td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td><strong>goal</strong></td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

* assumes we know location of node to be deleted
Binary Tree in C

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

Represent with TWO links per node.
- Left arrow for first link.
- Right arrow for second link.

Binary Search Tree

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

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

root (middle value)
left subtree (smaller values)
right subtree (larger values)

```
51
/   \
14    72
/     / \    /  \   /   /   \
66    NULL 06 NULL 33 53 97
```

Binary Search Tree

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

```
51
/   \
14    72
/     / \    /  \   /   /   \
06 13 25 43 64 84 99
```

Binary Search Tree

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

Binary tree in "sorted" order.
- Many BSTs correspond to 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 root 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 \( k \).

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

```c
Item BSTsearch(Key k) {
    return search(root, k);
}
```

Cost of BST Search

Cost of BST search depends on tree shape.
- Proportional to length of path from root to Key.
- If "balanced"
  - \( \log_2 N \) keys are compared
  - proportional to binary search cost
- But if "unbalanced"
  - takes \( N \) comparisons for degenerate tree shapes
  - can be as slow as sequential search

Tree shape depends on insertion method.
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

```c
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);
}
```

STbst.c (Sedgewick 12.7)

```c
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;
}
```

```
STbst.c (Sedgewick 12.7)
```

Allocate memory and initialize.

Insertion Cost in BST

Cost of BST insertion 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

With cleverness can ensure tree is always (sufficiently) balanced.
- See red-black trees in COS 226.
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>sorted array</td>
<td>search insert delete</td>
</tr>
<tr>
<td>log N insert N delete</td>
<td></td>
</tr>
<tr>
<td>instant 2 hour 2 hour</td>
<td></td>
</tr>
<tr>
<td>unsorted array</td>
<td>search insert delete</td>
</tr>
<tr>
<td>N insert N delete</td>
<td></td>
</tr>
<tr>
<td>2 hour instant instant</td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>search insert delete</td>
</tr>
<tr>
<td>log N insert log N delete</td>
<td></td>
</tr>
<tr>
<td>instant instant instant</td>
<td></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>
typedef char Key[30];
typedef struct {
    int ID;
    Key name;
} Item;
Item NULLItem = {-1, ""};
int eq(Key k1, Key k2) {
    return strcmp(k1, k2) == 0;
}
int less(Key k1, Key k2) {
    return strcmp(k1, k2) < 0;
}
int eq(Key, Key);
int less(Key, Key);
Key key(Item item) {
    return item.name;
}
```

Other Binary Trees

Tree properties
- Not all binary trees are BST’s.

Examples.
- Family (ancestor) tree.

```
Charles
  +
 /|
/ |
dad mom
Philip                        Elizabeth II
  +
 /|
/ |
Andrew Alice George VI Elizabeth
  +
 /|
/ |
George I Olga Louis Victoria George V Mary Claude Celia
```

Tree properties.
- Not all binary trees are BST’s.
- Order of children may not be important.

Examples.
- Family tree.
- Parse tree.
\[(a \ast (b + c)) + (d + e)\]
Other Types of Trees

Tree properties.
- Not all binary trees are BST’s.
- Order of children may not be important.
- Nodes need not have exactly two children.

Examples.
- Family (ancestor) tree.
- Parse tree.
- Unix file hierarchy.
- Descendant tree.

Summary

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

Performance characteristics using different data structures.

Binary tree is fundamental data structure in computer science.

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);
}

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

STbst.c

wrapper function

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

inorder

process node h

traverse left subtree

traverse right subtree
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

### inorder

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

### preorder

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

### postorder

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

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.

```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);
    }
}
```