Lecture P9: Trees

Overview

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

Trees
- Versatile and useful data structure.
- A naturally recursive data structure.
- 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 Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1920342006</td>
<td>Alam</td>
</tr>
<tr>
<td>2012121991</td>
<td>Baer</td>
</tr>
<tr>
<td>2021230087</td>
<td>Bagyenda</td>
</tr>
<tr>
<td>1779999898</td>
<td>Balestria</td>
</tr>
<tr>
<td>2328761212</td>
<td>Benjamin</td>
</tr>
<tr>
<td>1229993434</td>
<td>Berube</td>
</tr>
</tbody>
</table>

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.
**Representing the Database Entries**

Define `Item.h` file to encapsulate generic database entry.
- Don't want to use internals of item type when we write database code.
  - want our insert and search code to work for any item type
  - ideally `Item` would be an ADT
- Key is field in search.

```c
#include "ITEM.h"

int eq(Key A, Key B) {
    return A == B;
}

int less(Key A, Key B) {
    return A < B;
}

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

void print(Item A) {
    printf("%9d %20s", A.ID, A.name);
}
```

**Symbol Table ADT**

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

```c
ST.h (Sedgewick 12.1)

Item STsearch(Key);
void STinsert(Item);
void STprint(void);
int STcount(void);
void STdelete(Item);
```

**Sorted Array Representation of Database**

Maintain array of Items.
- Store in sorted order.
- Use BINARY SEARCH to find database Item with designated Key.

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

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

**Sorted Array Representation of Database**

Maintain array of Items.
- Store in sorted order.
- Use BINARY SEARCH to find database Item with designated Key.

```c
STarray.c (Sedgewick 12.6)

"Wrapper" for search function.

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

How many “comparisons” to find a name in database of size N?

- $\lfloor \log_2 N \rfloor$ = number of digits in binary representation of N.
- Divide list in half each time.

\[
\begin{align*}
5000 & \Rightarrow 2500 & \Rightarrow 1250 & \Rightarrow 625 & \Rightarrow 312 & \Rightarrow 156 & \Rightarrow 78 & \Rightarrow 39 \\
& \Rightarrow 18 & \Rightarrow 9 & \Rightarrow 4 & \Rightarrow 2 & \Rightarrow 1
\end{align*}
\]

The log functions grows very slowly.

- log₂ (thousand) = 10
- log₂ (million) = 20
- log₂ (billion) = 30

Without binary search (or if unsorted), may need to look at EVERY Item.

- Savings is enormous for large files.

Insert Using Sorted Array Representation

Problem 1: 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.

```
14 20 26 32 47 NULL

Demo: inserting 25 into a sorted array.
```

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.
**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.

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

**STlist.c**

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

**Summary**

Database entries.
- Names and social security numbers.

Desired operations.
- Insert, delete, search.

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

Tradeoff.
- ARRAY: fast search, slow insert/delete.
- LINKED LIST: fast insert/delete, slow search.

**Binary Tree**

Yes. Use TWO links per node.

```
root
  -- 14 --
     |
     v
  43

14
  -- 64 --
     |
     v
13

14
  -- 06 --
     |
     v
  33

14
  -- 64 --
     |
     v
  51
```

Is there any way to have fast insert AND search?
Binary Tree in C

Represent in C with two links per node.
- Leftmost arrow corresponds to left link
- Rightmost to right link.

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

Binary Search Tree

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

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

Binary Search Tree

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

- 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 > $k$.

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 assure 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.
Insert Using BST’s

```c
#include <string.h>

int eq(Key A, Key B) {
    return strcmp(A, B) == 0;
}

int less(Key A, Key B) {
    return strcmp(A, B) < 0;
}

Key key(Item A) {
    return A.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);

BST.c (Sedgewick 12.7)
link insert(link h, Item item) {
    Key k = key(item);
    Key k2 = key(h->item);
    link x;
    if (h == NULL) {
        link x = malloc(sizeof *x);
        x->item = item;
        x->left = x->right = NULL;
        return x;
    }
    if (less(k, k2))
        h->left = insert(h->left, item);
    else h->right = insert(h->right, item);
    return h;
}

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

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
  - With cleverness, can ensure tree is always balanced.
    - see COS 226

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
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.
Other Types of Trees

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

Examples.
- Family tree.
- Parse tree.
  \((a \times (b + c)) - (d + e)\)

Traversing Binary Trees

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

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

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

“Tree traversal.”
- Goal realized no matter what order nodes are visited.
  – inorder: visit between recursive calls

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

```
inorder
```
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

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

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

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

Important note: inorder traversal of BST gives free sort!

Preorder Traversal With Explicit Stack

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

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

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

Works for general trees. Generalizes to DEPTH-FIRST-SEARCH in graphs.
Level Traversal With Queue

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

level traversal with queue

```c
traverse(link h) {
    QUEUEput(h);
    while (!QUEUEempty()) {
        h = QUEUEget();
        print(h->item);
        if (h->left != NULL)
            QUEUEput(h->left);
        if (h->right != NULL)
            QUEUEput(h->right);
    }
}
```


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.