COS 126 Lecture 10: Trees

Class list

192-034-2006 Alam
201-212-1991 Baer
202-123-0087 Bagyenda
177-999-9898 Balestri
232-876-1212 Benjamin
122-999-3434 Berube
...

Desired operations

- add student
- return name, given ID number

SEARCH KEY

Similar applications

- online phone book
- airline reservations
- "symbol table"
- ...

GOAL: fast search *and* insert
even for huge databases

Array Representation: binary search

Define "Item.h" file to encapsulate item type

```
typedef int Key;
typedef struct{ Key ID; char name[30]; } Item;
#define eq(A, B) ((A.ID) == (B.ID))
#define less(A, B) ((A.ID) < (B.ID))
```

Keep array of Items, in sorted order

Use bisection method to find Item sought

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

Ex: search for 25

```
.   0  1  2  3  4  5  6  7  8  9 10 11 12
.  06 13 14 25 33 43 51 53 64 72 84 97 99
.  06 13 14 25 33 43
.           25 33 43
.           25
```

[See also Lecture 9; Programs 2.2 and 12.6]

Cost of Binary Search

Q: How many "comparisons" to find a name?

A: log N

divide list in half each time

Ex: 5000 -> 2500 -> 1250 -> 625 -> 312 -> 156 -> 78 -> 39 -> 18 -> 9 -> 4 -> 2 -> 1

log N = number of digits in decimal rep. of N

lg N = number of digits in binary rep. of N

lg(thousand) = 10

lg(million) = 20

lg(billion) = 30

Without binary search, might have to look at everything, so savings is substantial for very large files.

Problem: insert operation is usually slow

Ex: to insert 49

.	0	1	2	3	4	5	6	7	8	9	10	11	12
.	06	13	14	25	33	43	51	53	64	72	84	97	99

have to move larger keys over one position

.	0	1	2	3	4	5	6	7	8	9	10	11	12	13
.	06	13	14	25	33	43	49	51	53	64	72	84	97	99

Linked List Representation

Keep items in a linked list

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

Advantage of linked representation

can insert just by changing links

(no need to "move" anything)

```
.  06
.     13
.        14
.           25
.              33
.                 43
.                    51
.                 49    53
.                          64
.                             72
.                                84
.                                   97
.                                      99
```

Problem: can't have fast search

ARRAY: fast search, slow insert

LINKED LIST: slow search, fast insert

Binary Search Tree

Use *two* links per node

```
typedef struct STnode* link;
struct STnode { Item item; link l, r; };
```

Think of keys printed in order, left to right
take middle name for top, or "root" node
build tree recursively
"l" points to tree for left half
"r" points to tree for right half

```
.                    51

.         14                     72

.   06          33          53          97

. .    13    25    43    .     64    84    99

.     .  .  .  .  .  .        .  .  .  .  .  .

.   06 13 14 25 33 43 51    53 64 72 84 97 99
```

NULL links at bottom: "no information here"

Search in Binary Search Trees

Start at "head", link to the root

    if current node has key sought, return
    go left if key < key in current node
    go right if key > key in current node

```
Item searchR(link h, Key v)
  {
    if (h == NULL) return NULLitem;
    if eq(v, key(h->item)) return h->item;
    if less(v, key(h->item))
      return searchR(h->l, v);
    else return searchR(h->r, v);
  }
Item STsearch(Key v)
  { return searchR(head, v); }
```

Search cost:

    same as binary search in arrays
    if tree is balanced

Algorithm works for *any* tree shape

    will be slow for some tree shapes

Insert cost?

Insertion in binary search trees

Search for key not in tree
    ends on a NULL pointer
    node "belongs" there
    make a node, link it into the tree

```
link insertR(link h, Item item)
  { Key v = key(item);
     if (h == NULL)
        return NEW(item, NULL, NULL);
     if less(v, key(h->item))
        h->l = insertR(h->l, item);
     else h->r = insertR(h->r, item);
     return h;
  }
void STinsert(Item item)
  { head = insertR(head, item); }
```

Tree shape depends on key insertion order

Sorted, reverse order, etc.
    degenerates tree into linked list
"Random" order
    search, insert take about 1.44 lg N steps

BST construction example

```
.     33
.                                 33
.                              25          72
.     33                 06             51
.  25                       13       43
.
.
.     33
.  25    72                       33
.                              25          72
.                        06             51    99
.     33                    13       43
.  25       72
.        51
.
.
.        33
.     25       72
.  06       51
.
.
.           33
.        25       72
.  06            51
.    13
.
```

Other types of trees

Need not have precisely two children
Order might not matter

"Family" tree

```
.                      me

.           mom                    dad

. mom’s mom  mom’s dad  dad’s mom  dad’s dad
```

Parse tree (a * (b + c)) – (d + e)

```
.                     -
.          *              +
.       a      +       d   e
.            b   c
```

UNIX directory hierarchy

```
.                         /

.  bin lib etc sys            u

.                    bms dlj jma   rs   sis

.                       book3 C hw java mail src

.                            0 1 2 3 4
```

Traversing Binary Trees

Goal: "visit" (process) each node in the tree

```
visit(link h)
  { printf("%d %s ", h->item.ID, h->item.name);
traverse(link h)
  {
    if (h != NULL)
      {
        traverse(h->l);
        visit(h);
        traverse(h->r);
      }
  }
```

Goal realized no matter what order
    the statements in the "if" are executed

Preorder: visit before recursive calls
Inorder: visit between recursive calls
Postorder: visit after recursive calls

IMPORTANT NOTE:
    inorder search provides "free" SORT
    in binary search trees!

Preorder traversal with a stack

Visit the top node on the stack
    push its children

```
traverse(link t)
  {
    STACKpush(t);
    while (!STACKempty())
      {
        t = STACKpop(); visit(t);
        if (t->r != z) STACKpush(t->r);
        if (t->l != z) STACKpush(t->l);
      }
  }
```

Note: inorder more complicated
    (makes one appreciate recursion)

Preorder traversal of tree on slide 5:

```
.   51 14 06 13 33 25 43 72 53 64 97 84 99
```

Stack contents:

```
.
.   14 06 13 33 25 43 72 53 64 97 84 99
.   72 33 33 72 43 72    97 97    99
.      72 72    72
```

Works for general trees

Generalizes to DEPTH-FIRST SEARCH in graphs

Level order traversal

Use a queue instead of a stack

```
traverse(struct node *t)
  {
    QUEUEput(t);
    while (!QUEUEempty())
      {
        t = QUEUEget(); visit(t);
        if (t->l != z) QUEUEput(t->l);
        if (t->r != z) QUEUEput(t->r);
      }
  }
```

Visits nodes in order of distance from root

Level order traversal of tree on slide 5:

```
.   51 14 72 06 33 53 97 13 25 43 64 84 99
```

Queue contents:

```
.
.   14 72 06 33 53 97 13 25 43 64 84 99
.   72 06 33 53 97 13 25 43 64 84 99
.      33 53 97 13 25 43 64 84 99
.         97 13 25 43 64 84 99
.              43 64 84 99
.                   99
```

Works for general trees

Generalizes to BREADTH-FIRST SEARCH in graphs