Lecture 16. Writing Efficient Programs

• Is n a prime?

    int isprime(int n) {
        if (n > 2) {
            int i, m = n/2;
            for (i = 2; i < m; i++)
                if (n%i == 0)
                    return 0;
            return 1;
        }
    }

    int main(int argc, char *argv[]) {
        int i;
        for (i = 1; i < argc; i++) {
            int n;
            sscanf(argv[i], "%d", &n);
            if (isprime(n))
                printf("%d is a prime\n", n);
            else
                printf("%d is not a prime\n", n);
        }
        return 0;
    }

• 2147483647 is a prime, but isprime takes 1073741823 iterations to check!
Use a Better Algorithm

• Observations:

  Need to check only odd integers

  If \( n = a \times b \), then either \( a \) or \( b \) must be \(< \sqrt{n} + 1 \)

```c
#include <math.h>

int isprime(int n) {
    if (n > 2 && n%2 != 0) {
        int i, m = sqrt(n) + 1;
        for (i = 3; i < m; i += 2)
            if (n%i == 0)
                return 0;
        return 1;
    }
}
```

% lcc isprime2.c
% a.out 2147483647
2147483647 is a prime  ≈23169 iterations

• Better algorithms make programs faster, not microscopic code hacks

• Programs must be fast enough, not necessarily as fast as possible

• Don’t sacrifice clarity for speed
Searching

- A small 'database' problem: Maintain a list of names; lookup 'queries,' adding the new names, if necessary

```c
int main(int argc, char *argv[]) {
    int i;
    char buf[128];

    ptr = emalloc(size*sizeof (char *));
    ptr[0] = NULL;
    while (scanf("%s", buf) == 1)
        lookup(buf);
    for (i = 1; i < argc; i++) {
        int k = lookup(argv[i]);
        printf("%d\t%s\n", k, argv[i]);
    }
    printf("\n");
    for (i = 0; ptr[i] != NULL; i++)
        printf("%d\t%s\n", i, ptr[i]);
    return 0;
}
```

% lcc -I/u/cs126/include lookup.c /u/cs126/lib/libmisc.a
% a.out drh appel <names
3525 drh
794 appel
...
14210 zzwang
Searching, cont’d

• We know a good algorithm for searching — binary search (see page 10-3)

```
int bsearch(char *x[], int lb, int ub, char *q) {
    if (lb <= ub) {
        int m = (lb + ub)/2;
        int cond = strcmp(x[m], q);
        if (cond < 0)
            return bsearch(x, m + 1, ub, q);
        else if (cond > 0)
            return bsearch(x, lb, m - 1, q);
        else
            return m;
    } else
        return -1;
}
```

• `ptr[0..count-1]` holds the names in ascending order; `ptr[count]` is NULL

```
int count = 0;
char **ptr;

int lookup(char *name) {
    int k = bsearch(ptr, 0, count - 1, name);
    if (k == -1)
        k = insert(strsave(name));
    return k;
}
```
Cost of Binary Search

- Counting *comparisons* — calls to `strcmp` in this version of `bsearch` — is a good measure of the cost of binary search.

- Each recursive call cuts the problem in *half*, so the cost to search *N* names is:

  \[ C_N = C_{N/2} + 1 = C_{N/4} + 1 + 1 = \ldots \]

  Suppose *N* = \(2^n\), then

  \[ C_{2^n} = C_{2^{n-1}} + 1 = C_{2^{n-2}} + 1 + 1 = \ldots = C_1 + 1 + \ldots + 1 = n \]

  \[ C_N = \log_2 N = \lg N \]

  Even for huge *N*, \(\lg N\) is small (conversely, even for small *n*, \(2^n\) is huge…)

<table>
<thead>
<tr>
<th><em>N</em></th>
<th>(\lg N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>(10^k)</td>
<td>(\approx 3.129 \times k)</td>
</tr>
</tbody>
</table>

- Bottom line: Binary search, and other \(\lg N\) algorithms, are *fast*. 
Inserting Names

• To keep the names in ascending order, insert (q)

Expands the array, if necessary

Slides $\text{ptr}[k..\text{count}-1]$ down into $\text{ptr}[k+1..\text{count}]$ where $\text{ptr}[k] > q$

Stores $q$ in $\text{ptr}[k]$, increments count and sets $\text{ptr}[\text{count}]$ to NULL

```c
int size = 1;
int insert(char *q) {
    int k;
    if (count + 1 >= size) {
        size *= 2;
        ptr = erealloc(ptr, size*sizeof (char *));
    }
    for (k = count; k > 0 && strcmp(ptr[k-1], q) > 0; k--)
        ptr[k] = ptr[k-1];
    ptr[k] = q;
    ptr[++count] = NULL;
}
```

• Oh oh… If the array holds $N$ names, insert could take $N$ comparisons
insert in Action

% echo P R I N C E T O N | a.out

the ‘hole’ moves over dimmed letters
# Binary Search Trees

- Different representations have different costs

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Insertion</th>
<th>Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>fast</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>Linked list</td>
<td>slow: $\approx N$</td>
<td>fast w/search</td>
<td>fast w/search</td>
</tr>
<tr>
<td>Slow w/o search</td>
<td></td>
<td>slow w/o search</td>
<td>slow w/o search</td>
</tr>
<tr>
<td>Binary tree</td>
<td>fast: $\approx \log N$</td>
<td>fast</td>
<td>fast</td>
</tr>
</tbody>
</table>

- In a binary search tree

```c
struct node {
    char *key;
    int info;
    struct node *left, *right;
};
```

Names in the left subtree are $<$ than the name in the root

Names in right subtree are $\geq$ the name in the root

Holds for any node in the tree
Searching in Binary Trees

- To search for \( q \) in a binary search tree, start with \( \text{tree} = \text{root} \)
  
  1. If \( \text{tree is NULL} \), the search fails — an important boundary condition
  2. If \( q < \text{tree->key} \), search the \textit{left} subtree
  3. If \( q > \text{tree->key} \), search the \textit{right} subtree
  4. \( q \) must be equal to \( \text{tree->key} \)

```c
struct node *search(struct node *tree, char *q) {
    if (tree != NULL) {
        int cond = strcmp(q, tree->key);
        if (cond < 0)
            return search(tree->left, q);
        else if (cond > 0)
            return search(tree->right, q);
        else
            return tree;
    } else
        return NULL;
}
```

- Cost of searching in \textit{balanced} binary trees is the same as for binary search in arrays — \( \lg N \)
- It’s possible to keep trees balanced during insertion; take COS 226, Data Structures, to find out how, and read R. Sedgewick, \textit{Algorithms in C}, Addison-Wesley, 1990 (used in COS 226)
Searching, cont’d

```c
int count = 0;
struct node *root = NULL;

int lookup(char *name) {
    struct node *p = search(root, name);
    if (p == NULL) {
        p = insert(root, NULL, strsave(name));
        p->info = count++;
    }
    return p->info;
}

int main(int argc, char *argv[]) {
    int i;
    char buf[128];
    while (scanf("%s", buf) == 1) lookup(buf);
    for (i = 1; i < argc; i++) {
        int k = lookup(argv[i]);
        printf("%d\t%s\n", k, argv[i]);
    }
    print(root);
    return 0;
}
```
Printing Trees

- Sorting is ‘free:’ Print the left subtree, print the key, print the right subtree

```c
void print(struct node *tree) {
    if (tree != NULL) {
        print(tree->left);
        printf("%d\t%s\n", tree->info, tree->key);
        print(tree->right);
    }
}
```

% `lcc -I/u/cs126/include lookup2.c /u/cs126/lib/libmisc.a`
% `echo P R I N C E T O N | a.out`

```
4  C
5  E
2  I
3  N
7  O
0  P
1  R
6  T
```

- Ways to traverse trees; ‘visit’ means ‘process the node,’ e.g., print its key

<table>
<thead>
<tr>
<th>Traversal Type</th>
<th>Visit</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preorder</td>
<td>visit</td>
<td>traverse left</td>
<td>traverse right</td>
</tr>
<tr>
<td>Inorder</td>
<td>traverse left</td>
<td>visit</td>
<td>traverse right (alá print)</td>
</tr>
<tr>
<td>Postorder</td>
<td>traverse left</td>
<td>traverse right</td>
<td>visit</td>
</tr>
</tbody>
</table>
Inserting in Binary Trees

- insert is like search, but it must remember parent nodes in order to set the left or right field

`null`

insert must also handle the empty tree, which occurs when parent is `null`
Inserting in Binary Trees, cont’d

struct node *insert(struct node *tree, struct node *parent, char *q) {
    if (tree != NULL) {
        if (strcmp(q, tree->key) < 0)
            return insert(tree->left, tree, q);
        else
            return insert(tree->right, tree, q);
    } else {
        struct node *p = emalloc(sizeof (struct node));
        p->key = q;
        p->left = p->right = NULL;
        if (parent == NULL)
            root = p;
        else if (strcmp(q, parent->key) < 0)
            parent->left = p;
        else
            parent->right = p;
        return p;
    }
}

int lookup(char *name) {
    struct node *p = search(root, name);
    if (p == NULL) {
        p = insert(root, NULL, strsave(name));
        ...
    }
}