IS 2610: Data Structures

Searching

March 29, 2004
A symbol table is a data structure of items with keys that supports two basic operations: *insert* a new item, and *return* an item with a given key.

**Examples:**
- Account information in banks
- Airline reservations
Symbol Table ADT

- Key operations
  - Insert a new item
  - Search for an item with a given key
  - Delete a specified item
  - Select the $k$th smallest item
  - Sort the symbol table
  - Join two symbol tables

```c
void STinit(int);
int STcount();
void STinsert(Item);
Item STsearch(Key);
void STdelete(Item);
Item STselect(int);
void STsort(void (*visit)(Item));
```
Key-indexed ST

- Simplest search algorithm is based on storing items in an array, indexed by the keys

```
static Item *st;
static int M = maxKey;
void STinit(int maxN)
{
    int i;
    st = malloc((M+1)*sizeof(Item));
    for (i = 0; i <= M; i++) st[i] = NULLItem;
}

int STcount()
{
    int i, N = 0;
    for (i = 0; i < M; i++)
        if (st[i] != NULLItem) N++;
    return N;
}

void STinsert(Item item)
{
    st[key(item)] = item;
}

Item STsearch(Key v)
{
    return st[v];
}

void STdelete(Item item)
{
    st[key(item)] = NULLItem;
}

Item STselect(int k)
{
    int i;
    for (i = 0; i < M; i++)
        if (k-- == 0) return st[i];
}

void STsort(void (*visit)(Item))
{
    int i;
    for (i = 0; i < M; i++)
        if (st[i] != NULLItem)
            if (k-- == 0) return st[i];
}
```
Sequential Search based ST

- When a new item is inserted, we put it into the array by moving the larger elements over one position (as in insertion sort)

- To search for an element
  - Look through the array sequentially
  - If we encounter a key larger than the search key – we report an error
Binary Search

- Divide and conquer methodology
  - Divide the items into two parts
  - Determine which part the search key belongs to and concentrate on that part
    - Keep the items sorted
    - Use the indices to delimit the part searched.

```c
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);
}
Item STsearch(Key v)
{
    return search(0, N-1, v);
}
```
Binary Search Tree

- NST is a binary tree
  - A key is associated with each of its internal nodes
  - Key in any node
    - is larger than (or equal to) the keys in all nodes in that node’s left subtree
    - is smaller than (or equal to) the keys in all nodes in that node’s right subtree
- What is the output of inorder traversal on BST?
void STinsert(Item item)
{
    Key v = key(item); link p = head, x = p;
    if (head == NULL)
    {
        head = NEW(item, NULL, NULL, 1); return;
    }
    while (x != NULL)
    {
        p = x; x->N++;
        x = less(v, key(x->item)) ? x->l : x->r;
    }
    x = NEW(item, NULL, NULL, 1);
    if (less(v, key(p->item))) p->l = x;
    else p->r = x;
}

link insertR(link h, Item item)
{
    Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if less(v, t)
        h->l = insertR(h->l, item);
    else h->r = insertR(h->r, item);
    (h->N)++; return h;
}

void STinsert(Item item)
{
    head = insertR(head, item);
}
BST Complexities

- Best and worst case heights
  - $\ln N$ and $N$

- Search costs
  - Internal path length is related to – search hit
  - External path length is related to – search miss

- $N$ random keys
  - Average: Insertion, Search hit and Search miss require about $2 \ln N$ comparisons
  - Worst case search: $N$ comparisons
Basic Rotations

Transformations to rearrange nodes in a tree

- Maintain BST
- Changes three pointers

```c
link rotL(link h)
{ link x = h->r; h->r = x->l; x->l = h;
  return x; }
link rotR(link h)
{ link x = h->l; h->l = x->r; x->r = h;
  return x; }
```
Balanced Trees

- BST – worst case is bad!!
- Keep trees balanced so that searches can be done in less than $\ln N + 1$ comparisons
  - Maintenance cost incurred!
- Splay trees (Self-adjusting)
  - Tree automatically reorganizes itself after each op
  - When insert or search for $x$, rotate $x$ up to root using “double rotations”
  - Tree remains “balanced” without explicitly storing any balance information
Splay trees

- Check two links above current node
  - ZIG-ZAG: if orientations differ, same as root insertion
  - ZIG-ZIG: if orientations match, do top rotation first (unlike bottom rotation in root insertion using basic rotations)
2-3-4 Trees

- Nodes can hold more than one key
  - 2-nodes: 1 key; two links
  - 3-nodes: 2 keys; three links
  - 4-nodes: 3 keys; four links
- A balanced 2-3-4 tree
  - Links to empty trees are at the same height
2-3-4 Trees

- How do you Search?
- Insert
  - Search to bottom for key
  - 2-node at bottom: convert to 3-node
  - 3-node at bottom: convert to 4-node
  - 4-node at bottom – *split*
- Whenever root becomes 4 node – split it into a triangle of three 2-nodes
Red black trees

- Represent 2-3-4 trees as binary trees
Hashing

- Save items in a key-indexed table
  - Index is a function of the key
- Hash function
  - function to compute table index from search key
- Collision resolution strategy
  - Algorithms and data structures to handle two keys that hash to the same index
  - One approach – use linked list
Hashing

- **Time-space complexity**
  - No space limitation
    - Any search can be done in one memory access
  - No time limitation
    - Use limited memory and do sequential search
  - Limitation on both
    - Hashing to balance
Hash function: $h$

- Given a hash table of size $M$
  - $h$(Key) is a value in $[0,\ldots, M]$
  - Ideally, for each input, every output should be equally likely

- Simple methods
  - Modular hash function
    - $h(K) = K \mod M$; choose $M$ as prime
  - Multiplicative and modular methods
    - $h(K) = (K\alpha) \mod M$; choose $M$ as prime
    - A popular choice is $\alpha = 0.618033$ (golden ratio)
Hash Function: $h$

- Strings of characters
  - $26^4 \approx .5$ Million 4-char keys
  - Table size $M = 101$

<table>
<thead>
<tr>
<th>Binary</th>
<th>01100001</th>
<th>01100010</th>
<th>01100011</th>
<th>01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
<tr>
<td>Dec</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
<tr>
<td>ascii</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

- $abcd$ hashes to 11
  - $0x61626364 \% 101 = 16338831724 \% 101 = 11$
- $dcba$ hashes to 57
- Collision is inevitable
**Hash function: h**

- **Horner’s method**
  - $0x61626364 = 256*(256*(256*97+98) + 99)+100$
  - $0x61626364 \mod 101 = 256*(256*(256*97+98) + 99)+100 \mod 101$
  - Can take mod after each op
    - $(256*97+98) \mod 101 = 84$
    - $(256*84+99) \mod 101 = 90$
    - $(256*90+100) \mod 101 = 11$
  - N add, multiply and mod ops

<table>
<thead>
<tr>
<th>Binary</th>
<th>01100001</th>
<th>01100010</th>
<th>01100011</th>
<th>01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
<tr>
<td>Dec</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
<tr>
<td>ascii</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

```c
int hash(char *v, int M) {
    int h = 0, a = 127;
    for (; *v != \0; v++)
        h = (a*h + *v) % M;
    return h;
}
```

Why 127 instead of 128?
Universal Hashing and collision

- **Universal function**
  - Chance of collision for two distinct keys for table size M is precisely $1/M$

- **How to handle the case when two keys hash to the same value**
  - Separate chaining
  - Open addressing –
    - linear probe
    - Double hashing
  - Dynamic hash – increase table size dynamically

```c
int hashU(char *v, int M)
{
    int h, a = 31415, b = 27183;
    for (h = 0; *v != '\0'; v++,
        a = a*b % (M-1))
        h = (a*h + *v) % M;
    return h;
}
```

Performs well in practice!
Separate Chaining

- A linked list for each hash address
  - M linked lists
- M much smaller than N
- Property 14.1: Number of comparisons
  - Reduced by factor of M
  - Average length of the lists is N/M
- Search the list
  - Unordered:
    - insert takes constant time
    - Search is proportional to N/M
Open Addressing

- Open addressing
  - $M$ is much larger than $N$
  - Plenty of empty table slots
  - When a new key collides find an empty slot
  - Complex collision patterns

- Linear Probing
  - When collision occurs, check (probe) the next position in the table
    - Wrap around the table to find an empty slot
### Linear Probing

- **Load factor**
  - $\alpha$ - fraction of the table positions that are occupied (less than 1)
  - Search increases with the value of $\alpha$
  - Search loops infinitely when $\alpha = 1$
  - Insert: $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$

<table>
<thead>
<tr>
<th>A</th>
<th>S</th>
<th>E</th>
<th>R</th>
<th>C</th>
<th>H</th>
<th>I</th>
<th>N</th>
<th>G</th>
<th>X</th>
<th>M</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>11</td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>
Double Hashing

- Avoid clustering using second hash
- Take hash function relatively prime to avoid from probe sequence to be very short
  - Make M prime
  - Choose second has value that returns values less than M
    - A useful second hash: \((k \mod 97) + 1\)

\[
\text{insert: } \frac{1}{1-\alpha} \\
\text{search: } \frac{1}{\alpha} \ln(1+\alpha)
\]