Abstract Data Types Revisited (yet again)

Symbol Table, Dictionary

- items with keys
- INSERT
- SEARCH

Searching = "implement a symbol table ADT"

Clients

- online dictionary
- compiler symbol table
- index of any kind
- "associative memory"
Items and keys

Symbol table ADT clients and implementations need:
- definition of item and key types
- comparison operation
- test if equal operation
- scan in and print out items

Solution: define ADT for items

Ex: interface for char key that is an item

```c
typedef char Item;
typedef char Key;
#define key(A) A
#define less(A, B)  (A < B)
#define eq(A, B)  ((A) == (B))
#define maxKey 256
#define NULLitem 0

Item ITEMrand(void);
int ITEMscan(Item *);
void ITEMshow(Item);
```

Can switch to integers, big records, pointers, ... without changing ST implementation
Symbol table ADT

Items with keys

Two basic operations
insert
search (find item with given key)

Generic operations common to many ADTs
create
count (generalizes "test if empty")
destroy (often ignored if not harmful)

Other operations
construct (batch inserts)
sort (often a byproduct)
delete
find kth largest (generalized PQ)
join
Basic symbol-table ADT interface

- void STinit(int);
- void STinsert(Item);
- Item STsearch(Key);
- int STcount

**successful search:** returns item having key sought
**unsuccessful search:** returns NULLitem

constraints and error conditions
- arg to STinit: max size
- STinit and STinsert should return "no room" code

symbol-table ADT
- separates ST client from ST implementation

item ADT
- separates both from item implementation
Symbol table ADT client example

DEDUP a file of integers (remove duplicates)

```c
#include <stdio.h>
#include "Item.h"
#include "ST.h"

void main()
{
    int v; Item item;
    STinit(maxN);
    while (scanf("%d", &v) == 1)
    {
        if (STsearch(v) != NULLItem) continue;
        key(item) = v;
        STinsert(item);
        printf("%d ", v);
    }
}
```
ST ADT design issues

Equal keys?

Where's the record?
- ST implementation (copied from client)
- Client (passes pointer to ST implementation)

Operations mix
- All inserts, then all searches?
- Other operations? (sort, delete, count, find largest)
- ST manipulation operations (create, destroy, join)

What is a symbol table?

**BASIC GOAL:** fast search and fast insert
static Item *st;
static int N;
void STinit(int maxN)
{ st = malloc((maxN)*sizeof(Item)); N=0; }
int STcount()
{ return N; }
void STinsert(Item item)
{ int j = N++; Key v = key(item);
  while (j>0 && less(v, key(st[j-1])))
  { st[j] = st[j-1]; j--; }
  st[j] = item;
}
Item STsearch(Key v)
{ int j;
  for (j = 0; j < N; j++)
  { if (eq(v, key(st[j]))) return st[j];
    if (less(v, key(st[j]))) break; }
  return NULLItem;
}
Binary search

Maintain ordered array
Do search in \( \log N \) steps by divide-and-conquer

```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 example, analysis

Ex: search for I

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>E</td>
<td>G</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

l    r   m

-----------
1    12   6
1    5   3
4    5   4
5    5

THM: Binary search takes \( \lg N \) steps

Proof: Worst-case number of steps equals

number of bits in binary representation of \( N \)

(they satisfy the same recurrence)

- \( C(2N) = C(N) + 1 \)
- \( C(2N+1) = C(N) + 1 \)

Problem: Insert still slow
Interpolation search

Use key value to estimate probe position

Ex: search for I

- \( \frac{(I-A)}{(X-A)} = \frac{(9-1)}{(24-1)} = 0.35 \)
- therefore divide at 0.35, not 0.5
- \( 0.35 \times 12 = 5 \) (find I with one probe)

. 0 1 2 3 4 5 6 7 8 9 10 11 12
. A A E G I M N O R S T X

THM: Interpolation search takes \( \lg \lg N \) steps

Proof: [difficult]

(less than 5 probes in practice, since \( 2^{2^5} = \) billions)

Caveats: nonrandom files, cost of calculation

Problem: Insert still slow
## ST implementations cost summary

<table>
<thead>
<tr>
<th></th>
<th>worst-case</th>
<th></th>
<th>average-case</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>search</td>
<td>insert</td>
<td>search</td>
</tr>
<tr>
<td>ordered</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
</tr>
<tr>
<td>list</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
</tr>
<tr>
<td>unordered</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>N/2</td>
</tr>
<tr>
<td>list</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search</td>
<td>N</td>
<td>lgN</td>
<td>N/2</td>
<td>lgN</td>
</tr>
<tr>
<td>interp search</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>lg1lgN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>lgN</td>
<td>lgN</td>
</tr>
<tr>
<td>red-black</td>
<td>lgN</td>
<td>lgN</td>
<td>lgN</td>
<td>lgN</td>
</tr>
<tr>
<td>randomized</td>
<td>(N)</td>
<td>(N)</td>
<td>lgN</td>
<td>lgN</td>
</tr>
<tr>
<td>hashing</td>
<td>(N)</td>
<td>(N)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary search trees (BST)

Fundamental data structure

Recursive definition

a BST is
  a null link
  OR
  an item and two BSTs

Achieves basic ST performance goal
- fast search and fast insert

Flexible
- accommodate more ops than search and insert
- accommodate algs that GUARANTEE \( \log N \) performance
- [stay tuned]
BSTs are links
LINKs are pointers to nodes
NODEs are
- item with key
- left link (BST for smaller keys)
- right link (BST for larger keys)
- count of number of nodes in BST (optional)

typedef struct STnode* link;
struct STnode
    { Item item; link l, r; int N };  
static link head, z;

Special links
- head: head pointer (points to root)
- z: pointer to tail node (null link)
**NEW**: function to create a node
- fills in fields from args
- returns a link to the node

```c
link NEW(Item item, link l, link r, int N)
{
    link x = malloc(sizeof *x);
    x->item = item;
    x->l = l; x->r = r; x->N = N;
    return x;
}
```

```c
void STinit()
{
    head = (z = NEW(NULLItem, 0, 0, 0));
}
int STcount()
{
    return h->N;
}
```

null link: pointer to a node whose item is NULLItem
empty BST: null link
Recursive BST search implementation

Code directly follows from BST definition

```c
Item searchR(link h, Key v)
{
    Key t = key(h->item);
    if (h == z) return NULLItem;
    if eq(v, t) return h->item;
    if less(v, t)
        return searchR(h->l, v);
    else return searchR(h->r, v);
}

Item STsearch(Key v)
{
    return searchR(head, v);
}
```
**BST insertion**

*Search, then insert*

*Simply (but tricky) recursive code*
  - like inserting into a linked list

```c
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 insertion example
Nonrecursive BST insertion

```
STinsert(Item item)
{
    Key v = key(item);
    link p = head, x = p;
    if (head == z)
    {
        head = NEW(item, z, z); return;
    }
    while (x != z)
    {
        p = x;
        if less(v, key(x->item)) x = x->l;
        else x = x->r;
    }
    x = NEW(item, z, z, 1);
    if less(v, key(p->item)) p->l = x;
    else p->r = x;
}
```

Need parent pointer to link in new node
Equivalent recursive version is simpler
BST construction

Diagram of BST construction showing the insertion of elements A, E, S, C, R, I, G, H, N, M, P, and X.
Cost of search in BSTs

One-to-one correspondence
- BST search
- Quicksort partitioning

Total cost of searching for all nodes
- \( C(N) = N + 1 + 2 \left( C(1) + C(2) + \ldots + C(N-1) \right) / N \)
- [same recurrence as Quicksort]
  \[ = 2N \ln N \]

THM: Search and insertion in BSTs is logarithmic
(average case)

Problem: worst case is linear (too slow)
- nodes in order: degenerates to linked list

Can we GUARANTEE logarithmic performance?
- [stay tuned]
**SORT:** traverse tree in inorder

```c
void sortR(link h, void (*visit)(Item))
{
    if (h == z) return;
    sortR(h->l, visit);
    visit(h->item);
    sortR(h->r, visit);
}

void STsort(void (*visit)(Item))
{
    sortR(head, visit);
}
```

Same cost as Quicksort (+ space for links)
Useful ST operation comes for free
BST PQ implementation

GENERALIZED PQ: find kth smallest
Implement by adding tree size to nodes

```c
Item selectR(link h, int k) {
    int t = h->l->N;
    if (h == z) return NULLItem;
    if (t > k ) return selectR(h->l, k);
    if (t < k ) return selectR(h->r, k-t-1);
    return h->item;
}
Item STselect(int k) {
    return selectR(head, k);
}
```

O(log N) cost, on average
Special case: find smallest
Delete arbitrary nodes? Join?
  • [algorithmic and ADT issues]
Deletion in BSTs

To delete a node at the bottom (A E L P X)
- remove it

To delete a node with one child (A A C G I M)
- pass the child up

To delete a node with two children (S E H N)
- find the “next” node (use right-left* OR left-right*)
- swap, then remove (reduces to A or B)

Problem: strategy clumsy, not symmetric
Serious problem: trees not random (!!)
BSTs by root insertion

Idea: Insert such that new node stays at root
Motivation: Faster search for recently inserted nodes
FYI: Nonrecursive BST root insertion

void BSTinsert(Item item)
{
    Key v = key(item); int lr;
    struct BSTnode **t, **u, *x;
    x = NEW(rec, z, z);
    if (head == z) return x;
    if (less(v, key(head->item)))
    { x->r = head; head = x; t = &x->l;
        x = x->r; lr = 0; }
    else
    { x->l = head; head = x; t = &x->r;
        x = x->l; lr = 1; }
    while (x != z)
    { if (less(v, key(x->item)))
        { if ( lr) { lr = !lr; *t = *u; t = u; }
            u = &x->l; x = *u; }
        else
        { if (!lr) { lr = !lr; *t = *u; t = u; }
            u = &x->r; x = *u; }
        *t = z;
        return head; }
}

Challenge for the bored: check this code
Conclusion for the rest of us:
• recursive implementations *are* simpler!
ROTATE operation in BSTs

Fundamental operation
- rearrange nodes in trees
- local transformation (two links)
- maintain BST order at every node

Right rotate

Left rotate
ROTATE implementations

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

Easier done than said

Change just three links
Recursive BST root insertion implementation

To insert at root
  - insert at root of subtree (recursively)
  - rotate to bring inserted key to root

Equivalent to
  - insert the node
  - use rotations to bring it up to the top

Simple recursive implementation
Recursive BST root insertion code

```c
link insertT(link h, Item x)
{
    Key v = key(x);
    if (h == z) return NEW(x, z, z, l);
    if (less(v, key(h->x)))
    { h->l = insertT(h->l, x); h = rotR(h); }
    else
    { h->r = insertT(h->r, x); h = rotL(h); }
    return h;
}
void STinsert(Item x)
{ head = insertT(head, x); }
```

Not much more complicated than standard insertion
Faster if searches are for recently inserted keys (typical)
Basis for advanced algorithms
Rotate to the root
Top-down BST construction (root insertion)
ADT design alternatives

Single-instance ADT
- implementations in book, lecture
- useful in most applications
- allows us to focus on algorithms

First-class ADT
- can have multiple instances
- can assign to variables
- can use as arguments and return values for functions
- implementation effort usually rewarded

Challenge: allow client to manipulate instance
  WITHOUT knowing details of implementation

Solution: pointers to unspecified structs

More details:
- Sedgewick, sections 4.8, 9.5
- COS 217
First-class symbol-table ADT

Basic operations

- initialize
- insert
- search
- select kth smallest
- delete
- join two STs
- test if empty
- visit items in order

Handles (pointers to unspecified structs)

- client may need multiple STs
- may wish to use handles for items
- for delete, client needs to handle recs
- for join, client needs to handle STs
SUMMARY

- use elementary methods for small cases
- binary search guarantees $\lg N$ steps for search
  (but requires $N$ steps for insertion)
- interpolation search can speed up search
  (still requires $N$ steps for insertion)
- BSTs achieve basic goal
  fast search and insert, average-case
- BSTs give other operations (select, sort) as a byproduct
- can manipulate BSTs with rotations

Next goal:

- ST implementation with $O(\lg N)$ GUARANTEE for all ops