COS 226 Lecture 7: Searching overview

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 tems

Solution: define ADT for items

Ex: interface for char key that is an item

```
.  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)

```
#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

Ordered-array ST implementation

```
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 lgN steps by divide-and-conquer

```
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

```
.    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
.              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

- (I-A)/(X-A) = (9-1)/(24-1) = .35
- therefore divide at .35, not .5
- .35*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

	worst-case		average-case	
	insert	search	insert	search
ordered				
array	N	N	N/2	N/2
list	N	N	N/2	N/2
unordered				
array	1	N	1	N/2
list	1	N	1	N/2
binary search	N	lgN	N/2	lgN
interp search	N	N	N/2	lglgN
BST	N	N	lgN	lgN
red-black	lgN	lgN	lgN	lgN
randomized	(N)	(N)	lgN	lgN
hashing	(N)	(N)	1	1

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

- accomodate more ops than search and insert
- accomodate algs that GUARANTEE lgN performance
- [stay tuned]

BST data types

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)

BST initialization

NEW: function to create a node

- fills in fields from args
- returns a link to the node

```
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;
  }
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

```
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

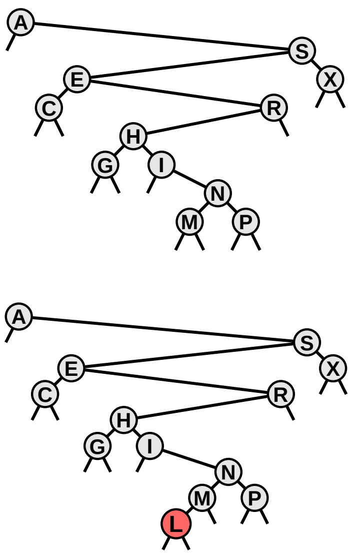
Search, then insert

Simply (but tricky) recursive code

- like inserting into a linked list

```
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



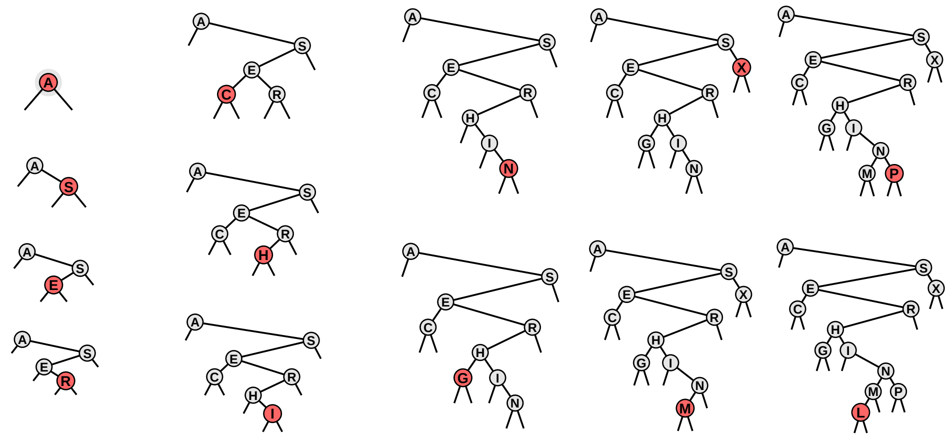
Nonrecursive BST insertion

```
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



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(C(1) + C(2) + ... + C(N-1))/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]

Other operations in BSTs

SORT: traverse tree in inorder

```
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

```
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)

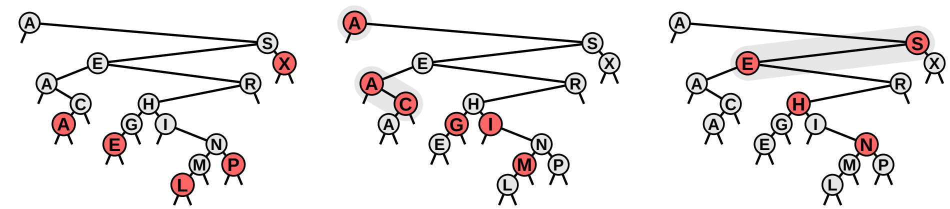
- remove it

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)



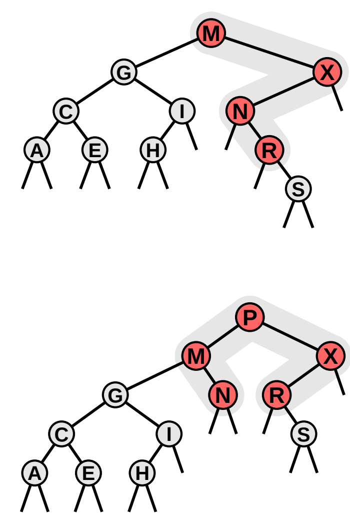
Problem: strategy clumsy, not symmetric

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:

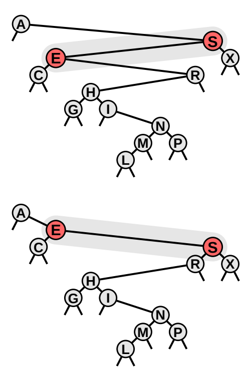
- recursive implementations *are* simpler!

ROTATE operation in BSTs

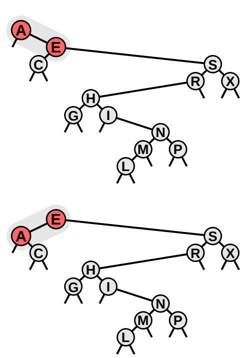
Fundamental operation

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

Right rotate



Left rotate



ROTATE implementations

```
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

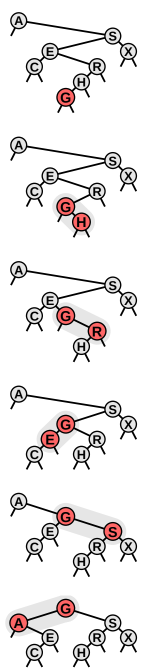
Simple recursive implementation

Recursive BST root insertion code

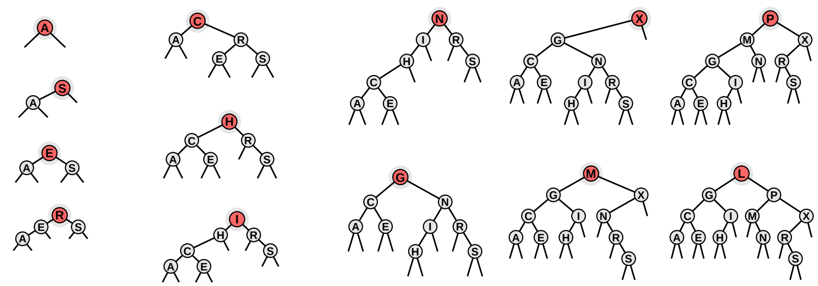
```
link insertT(link h, Item x)
  { Key v = key(x);
    if (h == z) return NEW(x, z, z, 1);
    if (less(v, key(h->x)))
    { h->l = insertT(h->l, x); h = rotR(h); }
    else
    { h->r = insertT(h->r, x); h = rotL(h); }
    return h;
  }
void STinsert(Item x)
  { head = insertT(head, x); }
```

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

Searching (to be continued)

SUMMARY

- use elementary methods for small cases
- binary search guarantees lgN 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(lgN) GUARANTEE for all ops