COS 226 Lecture 7: Searching overview

Abstract Data Types Revisited

Symbol Table, Dictionary

- records with keys
- INSERT
- SEARCH

Searching = "implement a symbol table ADT"

ADT orientation integral to modern languages
implementation in C possible

Records and keys

ADT for use by clients and implementations
of symbol table ADTs

Sample INTERFACE for char records and keys

```
.  typedef char keyType;
.  typedef char recordType;
.  #define nullrecord 0
.  #define key(A) A
.  #define less(A, B)  (A < B)
.  keyType randKey();
.      int scanKey(keyType *);
.     void showKey(keyType);
```

Client programs

use functions to create and write keys

manipulate records

Search algorithms (ST implementation)

store and retrieve records

use "key", "less" to extract, compare keys

Sample IMPLEMENTATION

```
char randKey()
  { return 'A' + 26*drand48(); }
int scanKey(char *x)
  { return scanf("%c", x); }
void showKey(char x)
  { printf(" %c ", x); }
```

"key" and "less" could be in implementation

Symbol table ADT

Records with keys

Two basic operations

* insert
* search (find record with given key)

generic operations common to many ADTs

* create
* test if empty
* destroy (often ignored if not harmful)

other operations

* sort (often a byproduct)
* delete
* find kth largest
* join

(ST > PQ)

INTERFACE for basic and generic operations

```
.       void STinit();
.       void STinsert(recordType x);
. recordType STsearch(keyType v);
.        int STempty();
```

Could combine with datatype interface

Should also specify

constraints and error conditions

Typical brute-force implementation

Use linear search in an ordered array

```
static recordType st[];
static int N;
STinsert(recordType rec)
   { int j = ++N;
      keyType v = key(rec);
      while (less(v, key(st[j-1])))
         {  st[j] = st[j-1]; j--; }
      st[j] = rec;
   }
recordType STsearch(keyType v)
   { int j = N+1;
      while (less(v, key(st[--j]))) ;
      if (v == key(st[j])) return st[j];
      return st[0];
   }
STinit() { N = 0; st[0] = nullrecord; }
int STempty() { return !N; }
```

Other options for brute-force implementations

* unordered array
* unordered singly-linked list
* ordered singly-linked list

Binary search

Maintain ordered array

Do search in lgN steps by divide-and-conquer

```
recordType STsearch(keyType v)
   { int l = 1; int r = N; int m;
      while (r >= l)
         {
            m = (l+r)/2;
            if (less(v, key(st[x]))) r = m-1; else l
            if (v == key(st[m])) return st[m];
         }
      return st[0];
   }
```

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
```

Worst-case number of steps for binary search =
number of bits in binary representation of N

Proof: satisfy the same recurrence

Interpolation search

Maintain ordered array

Improve upon binary search by using key values
to guess where to look

Ex: search for I
(I−A)/(X−A) = (9−1)/(24−1) = .35
therefore divide at .35, not .5
next probe will be on or close

.	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	5

THM: Interpolation search takes lglgN steps
for random files

Proof: difficult

Less than 5 probes in practice (2^2^5 = billions)

Caveats:
files not random
calculations cost more than comparisons

ST implementations cost summary

Per-operation time costs as function of ST size

	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
BST	N	N	lgN	lgN
red-black	lgN	lgN	lgN	lgN
randomized	(N)	(N)	lgN	lgN
hashing	N	N	1	1

Situation further complicated by

* need to have many other operations
* operations mix
* space costs

Another algorithm design success story:

* nearly optimal worst-case cost
* fast average-case cost
* simple (but ingenious!) algorithms

Full symbol table ADT

Basic operations

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

Handles

* for delete, client needs to handle recs
* for join, client needs to handle STs

INTERFACE for basic operations

```
.         ST STinit();
.   SThandle STinsert(ST, recordType);
. recordType STsearch(ST, keyType);
. recordType STselect(ST, int);
.       void STdelete(SThandle);
.         ST STjoin(ST, ST);
.        int STempty(ST);
.       void STvisit(ST);
```

Also could have search and select return handles

Symbol table ADT client example

Sort program

```
#include <stdio.h>
#include "types.h"   /* data types    */
#include "STtypes.h" /* handle types  */
#include "ST.h"      /* ADT interface */

void sort()
 {  keyType x;
     ST table = STinit();
     while (scanKey(&x) != EOF)
        table = STinsert(table, x);
     STvisit(table);
 }
```

Goal: independence from implementation

Space costs: same options as for PQs
* copy of data for implementation
* implementation manipulates data via indices
* implementation manipulates pointers

Examples of more complicated clients:
- online dictionary
- compiler symbol table

ADT implementation example

Need "interface" between abstract ADT definition and concrete implementation

- specify types of handles
- specify data type
- specify how to add new data
- (rename routines to keep client pure)

Ex: Binary search trees

```
typedef struct BSTnode* ST;
typedef struct BSTnode* SThandle;
#define z NULL
struct BSTnode
  { recordType rec; struct BSTnode *l, *r; };
SThandle
  NEW(recordType rec, SThandle l, SThandle r);
#define STinsert BSTinsert
#define STinit BSTinit
#define STvisit BSTvisit
```

interface to concrete ST implementations included in ST clients before abstract interface

Providing true "abstract" data type challenging

* it is worthwile to precisely specify ADT
* new implementations are "plug-ins"
* compiler type-checking can be helpful

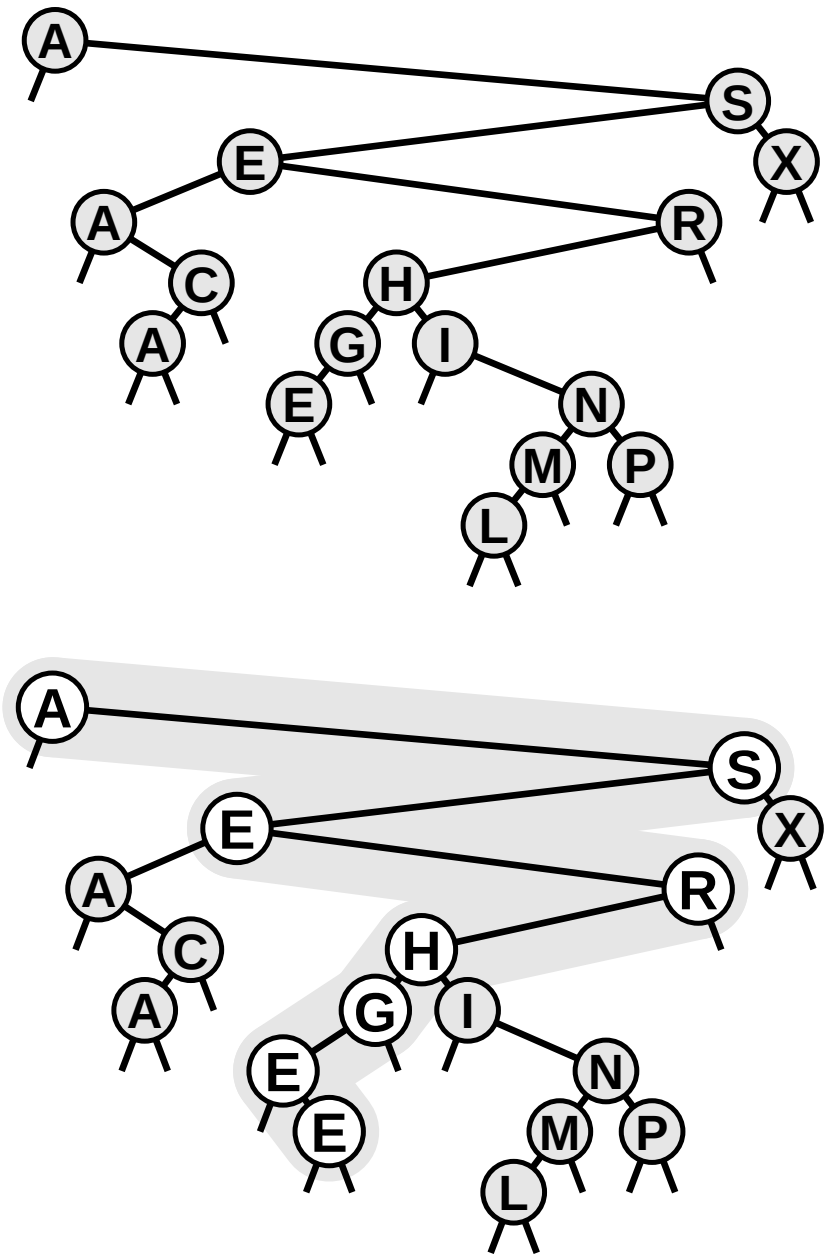
Binary search tree code

Recursive implementations are straightforward

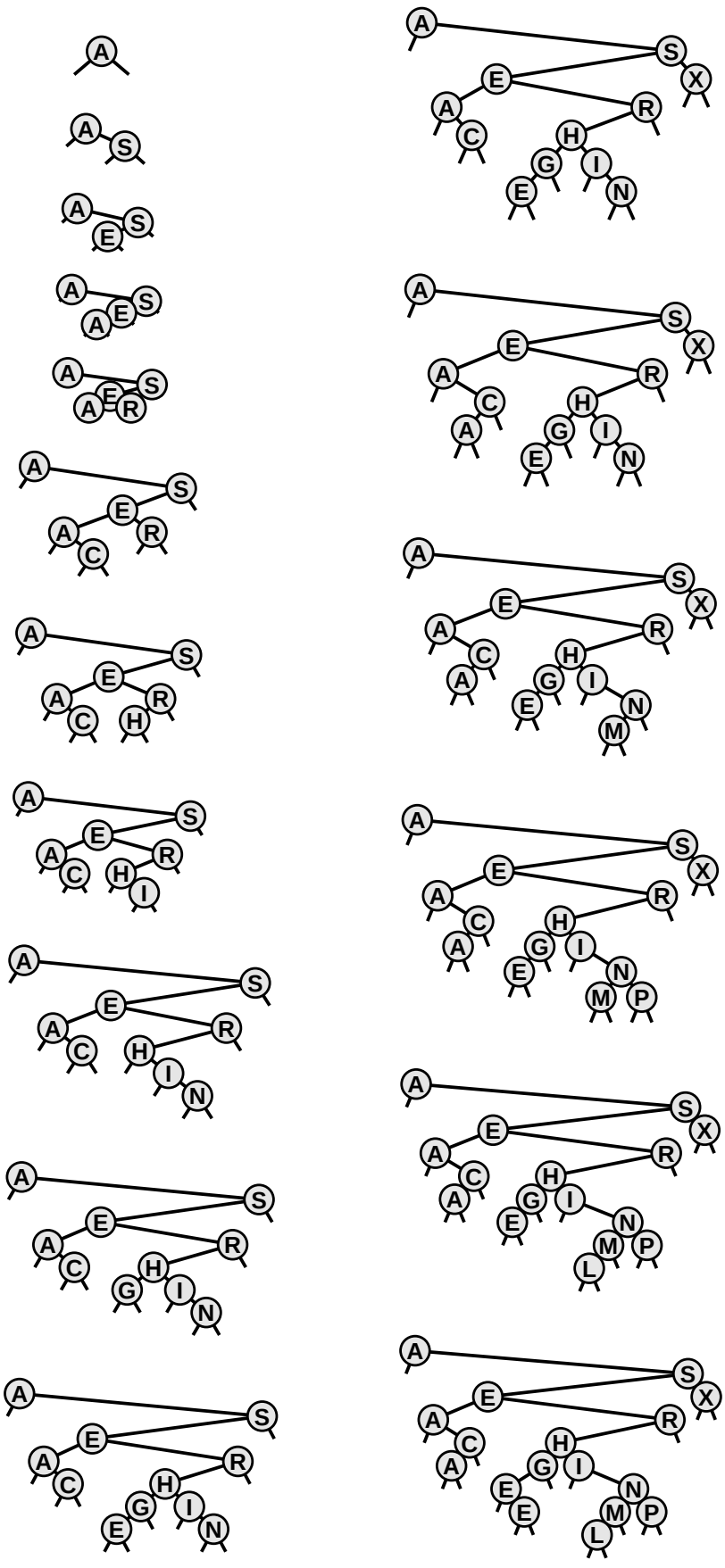
```
recordType BSTsearch(ST head, keyType v)
   {
      if (head == z) return nullrecord;
      if (v == key(head->rec)) return head->rec;
      if (less(v, key(head->rec)))
         return BSTsearch(head->l, v);
      else
         return BSTsearch(head->r, v);
   }
ST BSTinsert(ST head, recordType rec)
   { keyType v = key(rec);
      if (head == z)
         return head = NEW(rec, z, z, 1);
      if (less(v, key(head->rec)))
         head->l = BSTinsert(head->l, rec);
      else
         head->r = BSTinsert(head->r, rec);
      (head->N)++;
      return head;
   }
```

Compare w/ nonrecursive implementations (in book)

Binary search tree insertion example



Binary search tree construction



Cost of search in BSTs

One-to-one correspondence
between BST and Quicksort partitioning

Total cost of searching for all nodes
satisfies same recurrence as Quicksort

$$C(N) = N+1 + 2(C(1) + C(2) + \ldots + C(N-1))/N$$
$$\ldots$$
$$= 2N \ln N$$

THM: Search and insertion in BSTs involves examining 2lnN nodes on the average.

Worst case: nodes in order
degenerates to linked list, insertion cost N

Other operations in BSTs

SORT: traverse tree in inorder

```
BSTvisit(struct BSTnode *x)
   {
      if (x == z) return;
      BSTvisit(x->l);
      showKey(key(x->rec));
      BSTvisit(x->r);
   }
```

same cost as Quicksort (+ space for links)

Useful as "free" ST operation

GENERALIZED PQ: add tree size to nodes

```
recordType BSTselect(ST head, int k)
   { int t;
      if (head == z) return nullrecord;
      if (head->l == z) t=0; else t=head->l->N;
      if (t == k) return head->rec;
      if (t > k )
         return BSTselect(head->l, k);
      if (t < k )
         return BSTselect(head->r, k-t-1);
   }
```

special case: go left for smallest

Delete arbitrary node? Join?

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 D G I M)

pass the child up

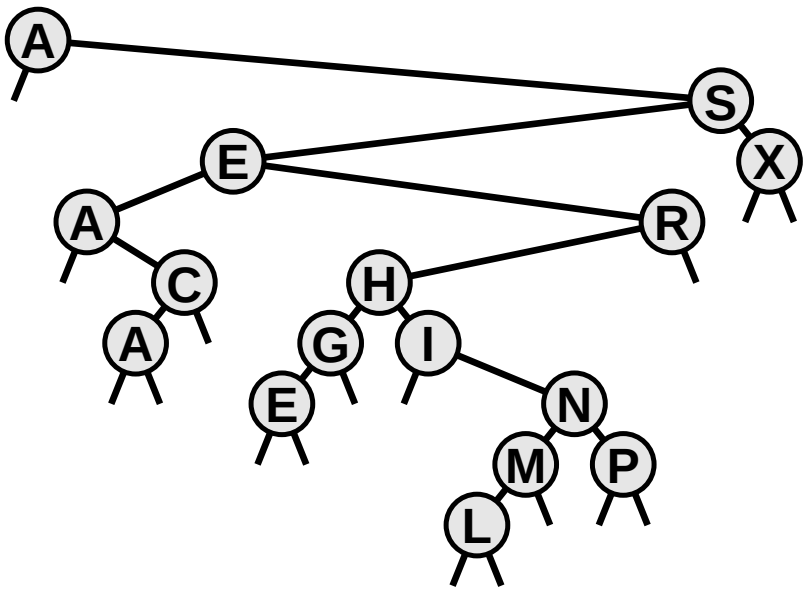
To delete a node with two children (S E H N)

find the "next" node

use right-left*

OR left-right*

swap, then remove (reduces to A or B)



Problem:

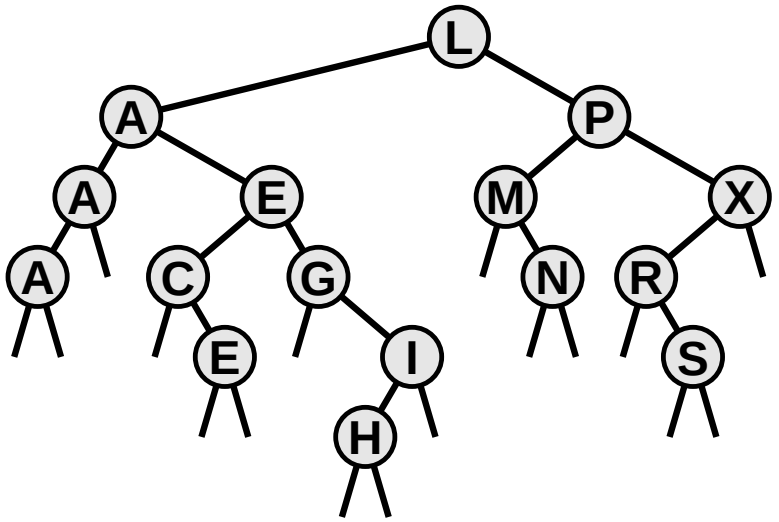
strategy not symmetric

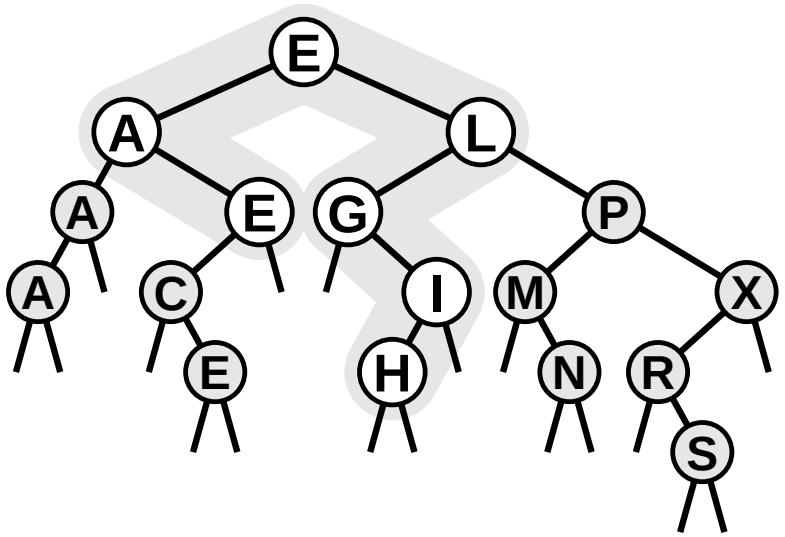
clumsy algorithm (see code in book!)

Serious problem: trees not random after long sequence of insertions and deletions (!!)

BSTs by root insertion

Idea: Insert such that new node stays at root





Can be done in one pass down

search for node value

pass up links that would cross

Faster search for recently inserted nodes

advantageous in some applications

BST root insertion code

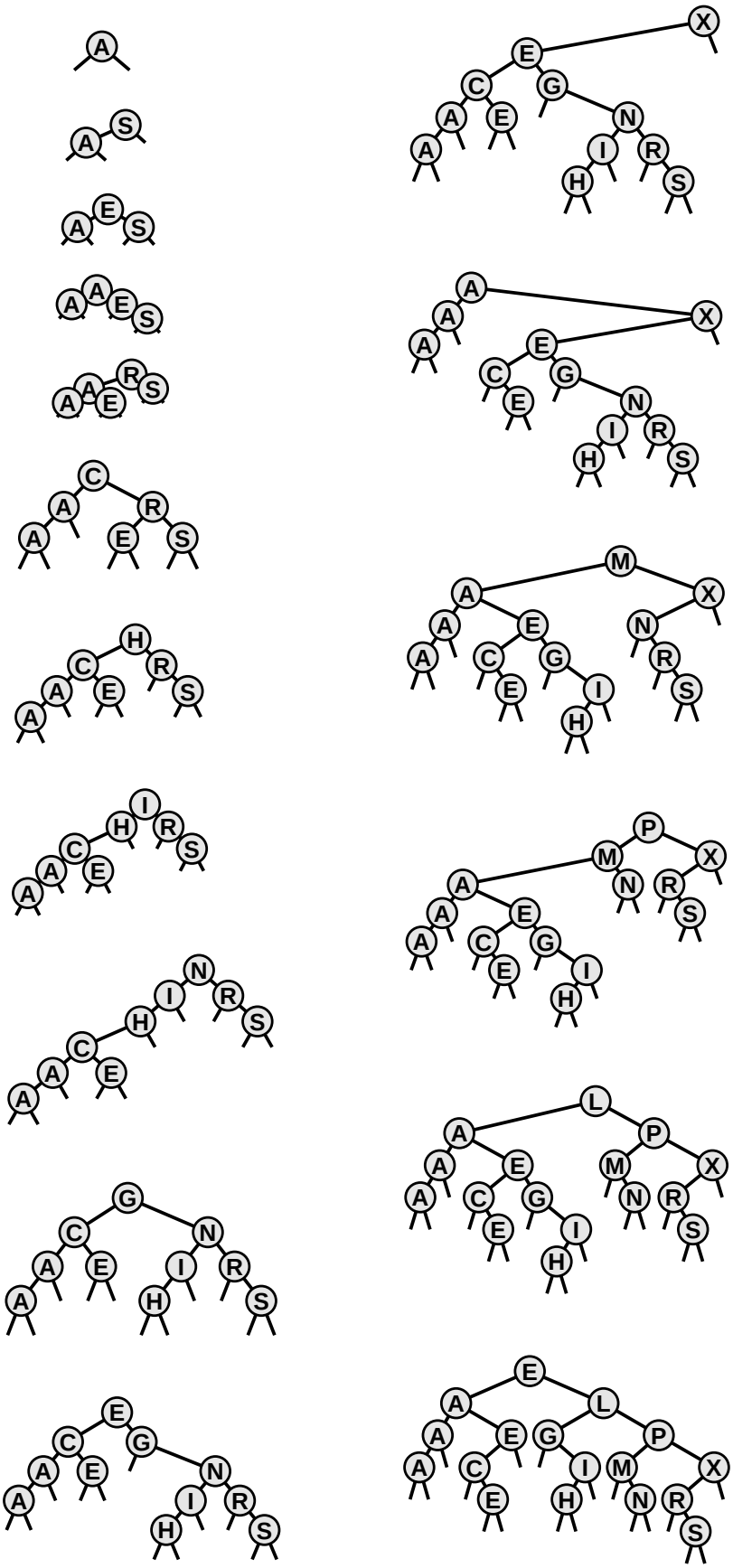
Easy idea, tricky code (but not much of it)

u: pointer to pointer of current node

t: pointer to pointer of last cross link

```
ST BSTinsert(ST head, recordType rec)
  { keyType v = key(rec); int lr;
    struct BSTnode **t, **u, *x;
    x = NEW(rec, z, z);
    if (head == z) return x;
    if (less(v, key(head->rec)))
      { 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->rec)))
      { 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;
  }
```

Top-down BST construction (root insertion)



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
  but requires N steps for insertion
* binary search trees give fast
  search and insert, average-case

Binary search trees can do more

* insert
* search
* delete
* select kth smallest
* ...

all in O(lgN) steps, average-case
with O(NlgN) sort as a byproduct

Goal: Symbol table implementation
with O(lgN) GUARANTEED performance
for both search and insert