Symbol Tables

These lecture slides have been adapted from:

- *Algorithms in C*, 3rd Edition, Robert Sedgewick.

Symbol Tables

Symbol table, dictionary.

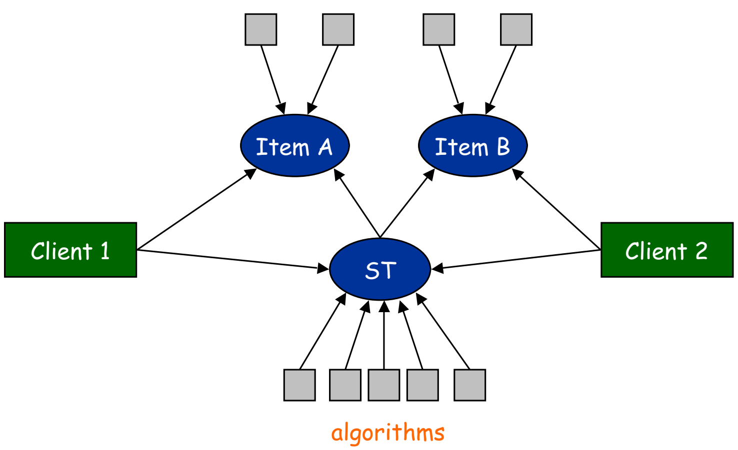
- Set of items with keys.
- INSERT a new item.
- SEARCH for an existing item with a given key.

Applications.

- Online phone book looks up names and telephone numbers.
- Spell checker looks up words in dictionary.
- Compiler looks up variable names to find type and memory address.
- Internet domain server looks up IP addresses.
- Web counter counts hits on web pages.
- "Associative memory."
- Index of any kind.

Abstract Data Types

Interface. Description of data type, basic ops.
Client. Program using ops defined in interface.
Implementation. Actual code implementing ops.



Key and Item Operations

Full set of Key operations.

- Create.
- Destroy.

generic ops for DT

- Compare.

ops that characterize Key

Full set of Item operations.

- Create.
- Destroy.
- Display.

generic ops for DT

- Extract key.

ops that characterize Item

```
typedef int Key;

typedef struct {
   Key ID;
   char name[30];
} Item;
```

Key = student ID
Item = ID + Name

Key and Item Operations

Full set of Key operations.

- Create.
- Destroy.

generic ops for DT

- Compare.

ops that characterize Key

Full set of Item operations.

- Create.
- Destroy.
- Display.

generic ops for DT

- Extract key.

ops that characterize Item

- Hit.

ops that specialized client might use

```
typedef char *Key;

typedef struct item *Item;
struct item {
   Key url;
   int count;
};
```

Key = URL
Item = URL + Count

Sample Item Interface

Item with a Key.

item.h

```
typedef char *Key;      // a string

typedef struct item *Item;
struct item {
   Key url;             // name of web page
   int count;           // number of hits
};

int eq(Key, Key);       // are keys equal?
int less(Key, Key);     // is 1st key smaller than 2nd?
int KEYscan(Key *);     // read in a Key from stdin

Key  ITEMkey(Item);     // extract key from item
Item ITEMinit(Key);     // init an Item
void ITEMshow(Item);    // print item to stdout
void ITEMhit(Item);     // increment number of hits
```


Sample Item Implementation

Item with a Key.

item.c (Key functions)

```
#include <stdio.h>
#include <stdlib.h>
#include "item.h"

#define MAXLEN 1000
char buffer[MAXLEN + 1];

int eq  (Key k1, Key k2) { return strcmp(k1, k2) == 0; }
int less(Key k1, Key k2) { return strcmp(k1, k2) <  0; }

int KEYscan(Key *pk) {
  int val = scanf("%1000s", buffer);
  *pk = malloc(strlen(buffer) + 1);
  strcpy(*pk, buffer);
  return val;
}
```


Sample Item Implementation

Item with a Key.

item.c (Item functions)

```
Key ITEMkey(Item a) { return a->url; }

void ITEMshow(Item a){
   printf("%s %d\n", a->url, a->count);
}

Item ITEMinit(Key k) {
   Item a = malloc(sizeof *a);
   a->count = 0;
   a->url = k;
   return a;
}

void ITEMhit(Item a) { a->count++; }
```

Symbol Table Operations

Set of Items with keys.

Full set of operations.
- Create.
- Destroy.

generic ops for ADT

- Insert.
- Search.

ops that characterize symbol table

- Count.
- Delete.
- Join.
- Sort.
- Find kth largest.

other ops that many clients need

```
Item item;
Key k;
STinit();
while(KEYscan(&k) == 1) {
   item = STsearch(k);
   if (item == NULL) {
      item = ITEMinit(k);
      STinsert(item);
   }
   ITEMhit(item);
   ITEMshow(item);
}
```

ST client that counts URL occurrences from input stream

Symbol Table: Unsorted Array Implementation

Maintain array of Items in arbitrary order.

INSERT: Add item at end of array.

55	32	47	6	4	82	26	56	20	14	58		
55	32	47	6	4	82	26	56	20	14	58	28	

Key = Item = int

SEQUENTIAL SEARCH: Iterate through all elements of array.

4	6	14	20	26	82	32	47	55	56	58	28	

Symbol Table: Unsorted Array Implementation

STunsortedarray.c

```
#define MAXSIZE 10000          // max # items
static Item st[MAXSIZE];       // array of items
static int N = 0;              // number of elements

Item STinsert(Item item) { st[N++] = item; }

Item STsearch(Key k) {
   int i;
   for (i = 0; i < N; i++)
      if eq(k, ITEMkey(st[i]))
         return st[i];          // found
   return NULL;                 // not found
}
```


Symbol Table: Sorted Array Implementation

Maintain array of Items sorted by `Key`.

BINARY SEARCH:
- Examine the middle Key.
- If it matches, then we're done.
- Otherwise, search either the left or right half.

INSERT: Find insertion point and shift items right.

4	6	14	20	26	32	47	55	56	58	82		
4	6	14	20	26	28	32	47	55	56	58	82	

Key = Item = int

Symbol Table: Sorted Array Implementation

STsortedarray.c (Sedgewick 12.6)

```
#define MAXSIZE 10000
static Item st[MAXSIZE];
static int N = 0;

static Item search(int lo, int hi, Key k) {
  int m = (lo + hi) / 2;
  if (lo > hi)                    // not found
     return NULL;
  else if eq(k, ITEMkey(st[m]))   // found
     return st[m];
  else if less(k, ITEMkey(st[m])) // go left
     return search(lo, m-1, k);
  else                            // go right
     return search(m+1, hi, k);
}

Item STsearch(Key k) { return search(0, N-1, k); }
```


Binary Search Analysis

How many comparisons to find a name in database of size N?

- Divide list in half each time.
- 625 ⇒ 312 ⇒ 156 ⇒ 78 ⇒ 39 ⇒ 18 ⇒ 9 ⇒ 4 ⇒ 2 ⇒ 1
- $625_{10} = 1001110001_2$

Theorem. Binary search takes O(log N) steps.

Proof. Worst-case number of steps satisfies:

- $C(N) = 1 + C(N/2)$ (integer division)
- $C(1) = 0$

  ⇒ $C(N) = \lceil \log_2 (N+1) \rceil$
- Same recurrence as # bits in binary representation of N.

Symbol Table: Implementations Cost Summary

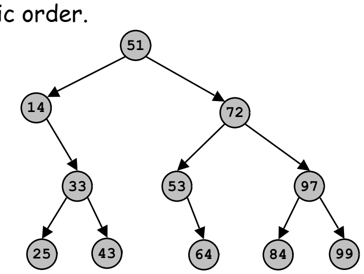
	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete *
Unsorted array	N	1	1	N/2	1	1
Sorted array	log N	N	N	log N	N/2	N/2

* assumes we know location of node to be deleted

Can we achieve log N performance for all ops?

Binary Search Tree

Binary search tree: binary tree in symmetric order.

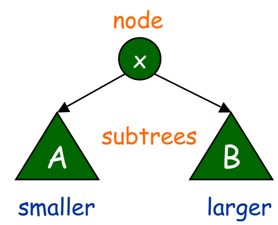


Binary tree is either:

- Empty (NULL node).
- An Item and two binary trees.

Symmetric order:

- Keys in nodes.
- No smaller than left subtree.
- No larger than right subtree.



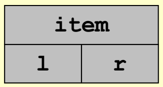
Binary Search Tree in C

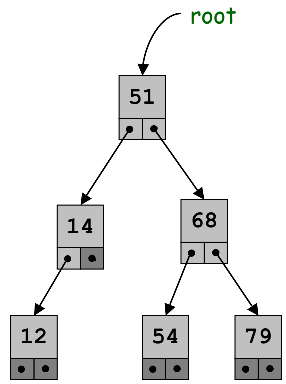
BST is a link.
LINK is a pointer to a node.
NODE is comprised of three fields:

- Item with a key.
- Left link (BST with smaller keys).
- Right link (BST with larger keys).

STbst.c

```
typedef struct STnode* link;
struct STnode {
   Item item;
   link l;
   link r;
};

static link root;
```

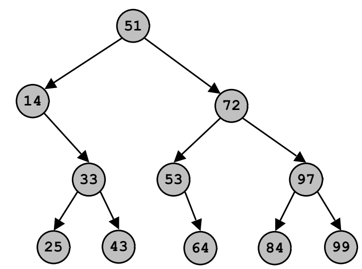


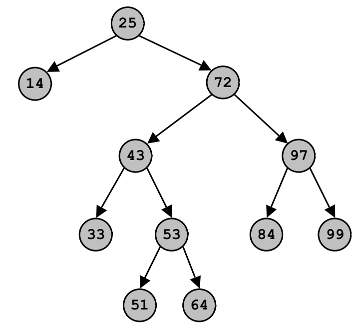


Tree Shape

Tree shape.

- Many BSTs correspond to same input data.
- Have different tree shapes.
- Performance depends on shape.





BST Search

Search for `Key k`.

- Code follows from BST definition.

STbst.c (Sedgewick 12.7)

```
Item search(link x, Key k) {
  if (x == NULL)                // not found
     return NULL;

  if (eq(k, ITEMkey(x->item))   // found
     return x->item;

  if (less(k, ITEMkey(x->item)) // go left
     return search(x->l, k);

  return search(x->r, k);       // go right

}

Item STsearch(Key k) { return search(root, k); }
```


BST Insert

Insert `Item item`.

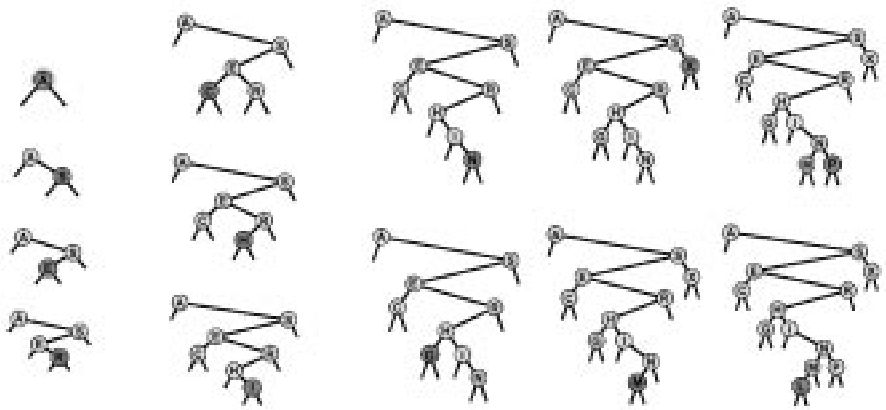
- Search, then insert.
- Simple (but tricky) recursive code.

STbst.c (Sedgewick 12.7)

```
link insert(link x, Item item) {
   if (x == NULL)                              // insert here
      return NEWnode(item, NULL, NULL);

   if (less(ITEMkey(item), ITEMkey(x->item)) // go left
      x->l = insert(x->l, item);

   else                                        // go right
      x->r = insert(x->r, item);

   return x;
}

void STinsert(Item item) { root = insert(root, item); }
```

BST Construction

Insert the following keys into BST: **A S E R C H I N G X M P L**



BST Analysis

Cost of search and insert BST.

- Proportional to depth of node.
- 1-1 correspondence between BST and quicksort partitioning.
- Height of node corresponds to number of function calls on stack when node is partitioned.

Theorem. If keys are inserted in random order, then height of tree is $\Theta(\log N)$, except with exponentially small probability. Thus, search and insert take $O(\log N)$ time.

Problem. Worst-case search and insert are proportional to N.

- If nodes in order, tree degenerates to linked list.

Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete *
Unsorted array	N	1	1	N/2	1	1
Sorted array	log N	N	N	log N	N/2	N/2
BST	N	N	N	log N	log N	???

* assumes we know location of node to be deleted
† if delete allowed, insert/search become sqrt(N)
‡ probabilistic guarantee

BST: log N insert and search IF keys arrive in RANDOM order.
Ahead: Can we make all ops log N if keys arrive in ARBITRARY order?

Other Symbol Table Operations

SORT: traverse tree in inorder.

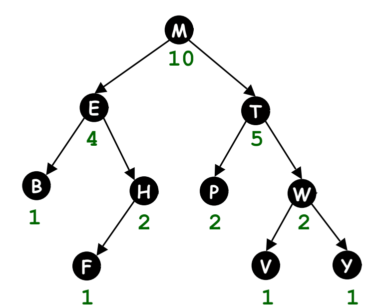
- Same cost as quicksort, but pay space for extra links.

FIND KTH: generalized priority queue that finds kth smallest.

- Special case: find min, find max.
- Add subtree size to each node.
- Takes time proportional to height of tree.

RANGE SEARCH.

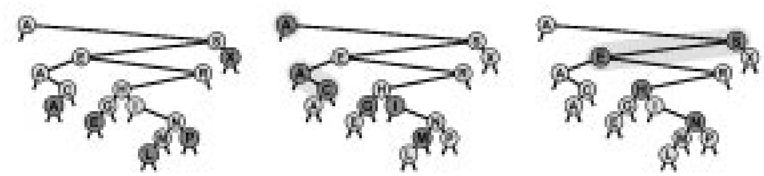
```
typedef struct node {
   link l, r;
   Item item;
   int N;
}
```



Other Symbol Table Operations: Delete

To delete at node:

- At the bottom ⇒ just remove it. A E L P X
- With one child ⇒ pass the child up. A A C G I M
- With two children ⇒ S E H N
  - find the next largest node using right-left* or left-right*
  - swap
  - remove as above since it now has zero or one children



Problem: strategy clumsy, not symmetric.
Serious problem: trees not random (!!)

Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete *
Unsorted array	N	1	1	N/2	1	1
Sorted array	log N	N	N	log N	N/2	N/2
BST	N	N	N	log N	log N	sqrt(N) †

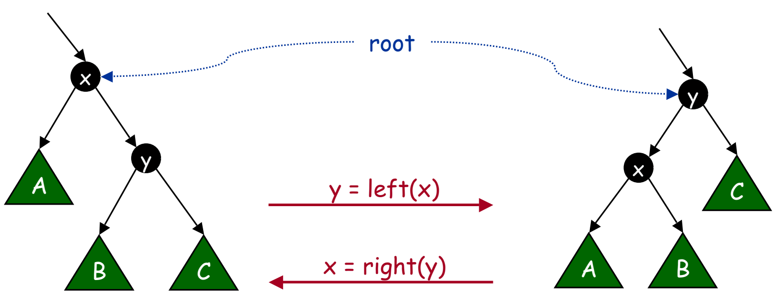
* assumes we know location of node to be deleted
† if delete allowed, insert/search become sqrt(N)

Ahead: Can we achieve log N delete?
Ahead: Can we achieve log N worst-case?

Right Rotate, Left Rotate

Fundamental operation to rearrange nodes in a tree.

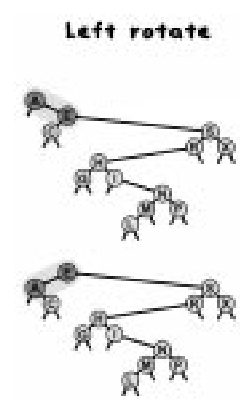
- Maintains BST order.
- Local transformations, change just 3 pointers.



Right Rotate, Left Rotate

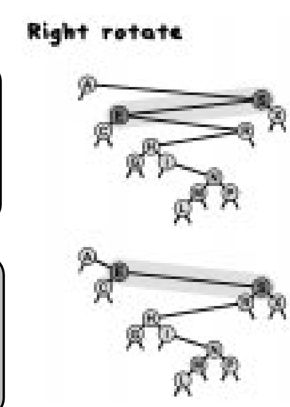
Fundamental operation to rearrange nodes in a tree.

- Easier done than said.




```
link rotL(link x) {
   link y = x->r;
   x->r = y->l;
   y->l = x;
   return y;
}
```

```
link rotR(link y) {
   link x = y->l;
   y->l = x->r;
   x->r = y;
   return x;
}
```



Recursive BST Root Insertion

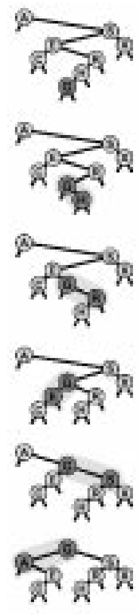
Root insertion: insert a node and make it the new root.

- Insert the node using standard BST.
- Use rotations to bring it up to the root.

Why bother?

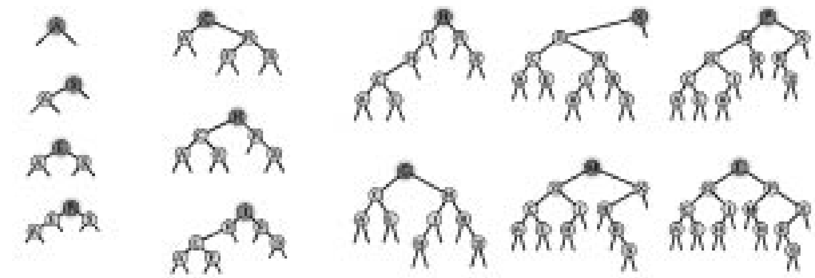
- Faster if searches are for recently inserted keys.
- Basis for advanced algorithms.

```
link insert(link x, Item item) {
    if (x == NULL) return NEW(item, NULL, NULL);
    if (less(ITEMkey(item), ITEMkey(x->item)) {
        x->l = insert(x->l, item);
  ➡    x = rotR(x);
    }
    else {
        x->r = insert(x->r, item);
  ➡    x = rotL(x);
    }
    return x;
}
```



BST Construction: Root Insertion

A S E R C H I N G X M P L



Randomized BST

Observation. If keys are inserted in random order then BST is balanced with high probability.

Idea. When inserting a new node, make it the root with probability 1/(N+1) and do it recursively.

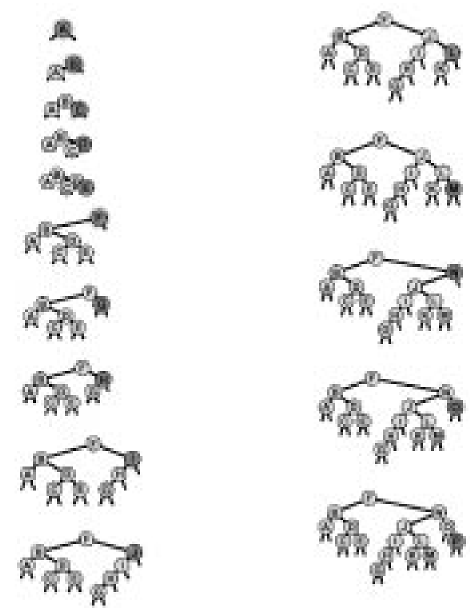
Fact. Tree shape distribution is identical to tree shape of inserting keys in random order.

- Choosing random partition element is analogous to having shuffled input before sorting.
- No assumptions made on the input distribution!

Randomized BST Example

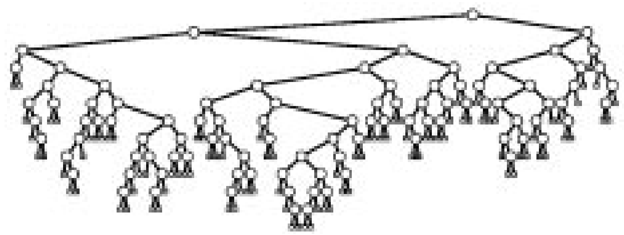
Insert keys in order.

- Tree shape still random.



Randomized BST

Always "looks like" random binary tree.



- Implementation straightforward.
  - maintain subtree size in each node
- Supports all symbol table ops.
- log N average case.
- Exponentially small chance of bad balance.

Randomized BST: Other Operations

FIND kth largest. Use subtree size field already stored.

JOIN two disjoint STs.

- To join two symbol tables A (of size M) and B (of size N):
  - use A as root with probability M / (M + N)
  - use B as root with probability N / (M + N)
  - join other tree with subtree recursively

DELETE a node. Delete the node. JOIN broken subtrees as above.

Theorem. Trees still random after delete (!!)

Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete *
Unsorted array	N	1	1	N/2	1	1
Sorted array	log N	N	N	log N	N/2	N/2
BST	N	N	N	log N	log N	sqrt(N) †
Randomized	log N ‡	log N ‡	log N ‡	log N ‡	log N ‡	log N ‡

* assumes we know location of node to be deleted
† if delete allowed, insert/search become sqrt(N)
‡ probabilistic guarantee

Randomized BST: guaranteed log N performance!
Next time: Can we achieve deterministic guarantee?