

## COS 226 Lecture 10: Radix trees and tries

### Symbol Table, Dictionary

- records with keys
- INSERT
- SEARCH

### Balanced trees, randomized trees

- use  $O(\lg N)$  comparisons

### Hashing

- uses  $O(1)$  probes
- but time proportional to key length

### Are comparisons necessary?

- (no)

### Is time proportional to key length required?

- (no)

**Best possible:** examine  $\lg N$  BITS

## Digital search trees (DSTs)

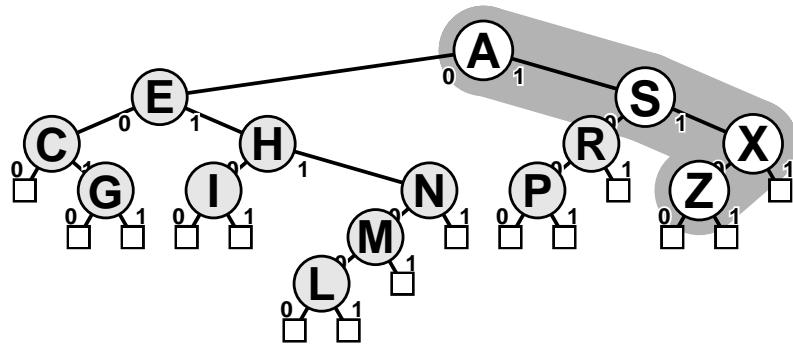
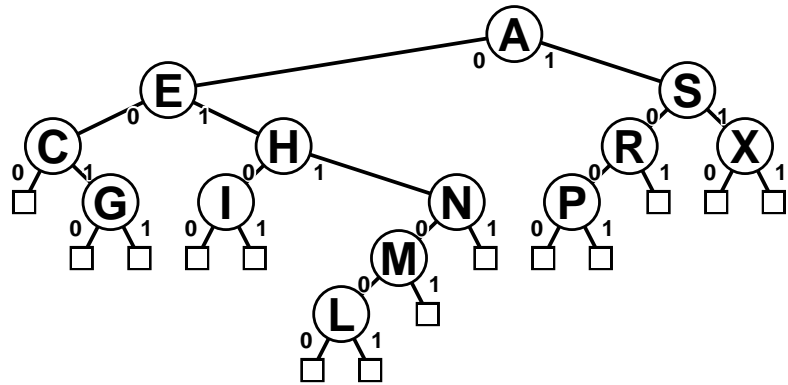
Easy way to balance tree: use bits of key to direct search  
Otherwise identical to BST

```
#define bit(A, B) digit(A, B)
Item searchR(link h, Key v, int w)
{ Key t = key(h->item);
  if (h == z) return NULLitem;
  if eq(v, t) return h->item;
  if (bit(v, w) == 0)
      return searchR(h->l, v, w+1);
  else return searchR(h->r, v, w+1);
}
Item STsearch(Key v)
{ return searchR(head, v, 0); }
```

“digit” macro extracts Bth bit from A (see lecture 6)

# Digital tree insertion

A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100



Trees NOT ordered

- does not support SORT and SELECT

## DST insertion code

```
link insertR(link h, Item item, int w)
{ Key v = key(item), t = key(h->item);
  if (h == z) return NEW(item, z, z, 1);
  if (bit(v, w) == 0)
      h->l = insertR(h->l, item, w+1);
  else h->r = insertR(h->r, item, w+1);
  return h;
}
void STinsert(Item item)
{ head = insertR(head, item, 0); }
```

## Tries

Branch according to bits in keys

No keys in internal nodes

Records/keys in external nodes

Structure depends on keys, not insertion order

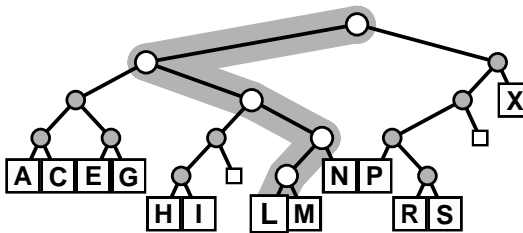
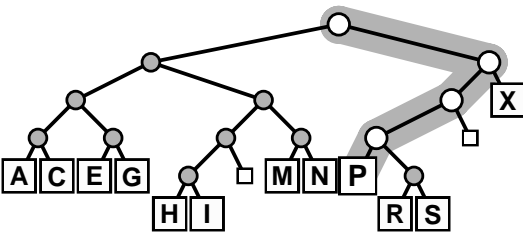
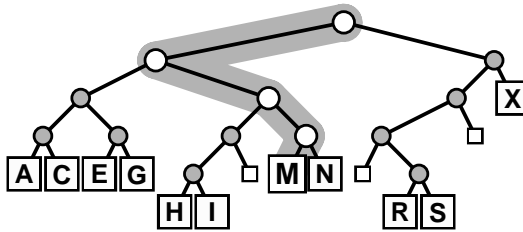
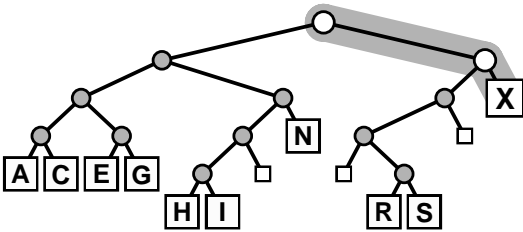
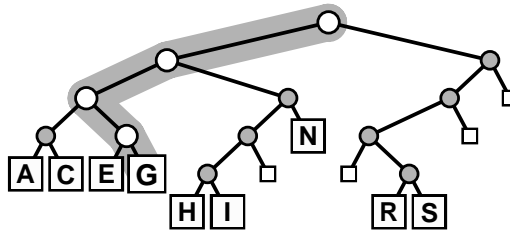
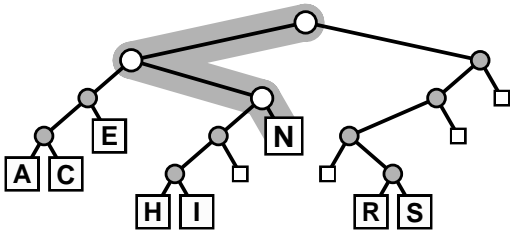
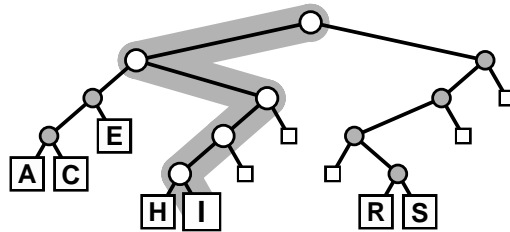
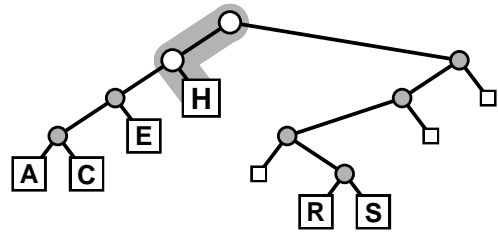
Examine  $\lg N$  BITS to distinguish key

- independent of key length!

### Problems:

- 44% space waste from 1-way branching
- multiple node types

# Trie example



## Trie search implementation

Branch according to bits, as in DST

Three outcomes:

- null link
- key in external node matches search key
- key in external node differs from search key

```
#define external(A) ((h->l == z) && (h->r == z))
Item searchR(link h, Key v, int w)
{
    Key t = key(h->item);
    if (h == z) return NULLitem;
    if (external(h))
        return eq(v, t) ? h->item : NULLitem;
    if (digit(v, w) == 0)
        return searchR(h->l, v, w+1);
    else return searchR(h->r, v, w+1);
}
Item STsearch(Key v)
{
    return searchR(head, v, 0);
}
```

## Trie insertion implementation

Two possible search miss outcomes

- null link: replace with link to new node
- external node: recursive split to distinguish new key

```
void STinit()
{ head = (z = NEW(NULLitem, 0, 0, 0)); }
link insertR(link h, Item item, int w)
{ Key v = key(item), t = key(h->item);
  if (h == z) return NEW(item, z, z, 1);
  if (external(h))
  { return split(NEW(item, z, z, 1), h, w); }
  if (digit(v, w) == 0)
    h->l = insertR(h->l, item, w+1);
  else h->r = insertR(h->r, item, w+1);
  return h;
}
void STinsert(Item item)
{ head = insertR(head, item, 0); }
```

## Trie insertion recursive node split

```
link split(link p, link q, int w)
{ link t = NEW(NULLitem, z, z, 2);
  switch(bit(p->item, w)*2+bit(q->item, w))
  { case 0: t->l = split(p, q, w+1); break;
    case 1: t->l = p; t->r = q; break;
    case 2: t->r = p; t->l = q; break;
    case 3: t->r = split(p, q, w+1); break; }
  return t;
}
```

# Patricia tries

Digression: cute nomenclature

**P** ractical

**A** lgorithm

**T** o

**R** etrieve

**I** nformation

**C** oded

**I** n

**A** lphanumeric

information reTRIEval (but pronounced "try")

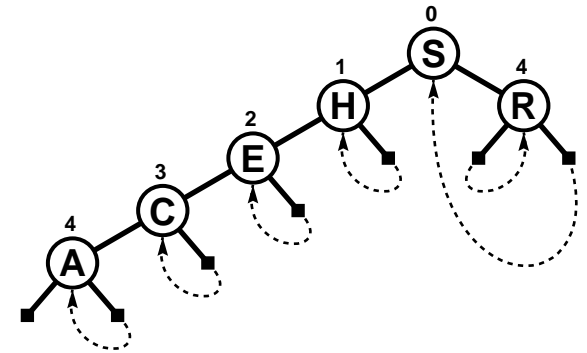
Patricia:

- collapse one-way branches in tries
- "thread" tree to eliminate multiple node types

Quintessential search algorithm

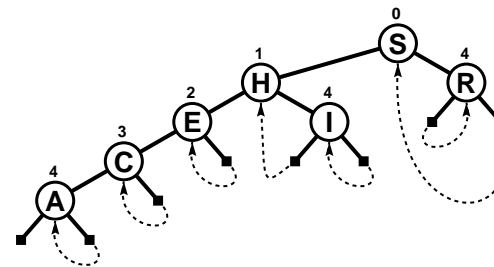
# Patricia trie search and insertion

**SEARCH:** branch according to specified

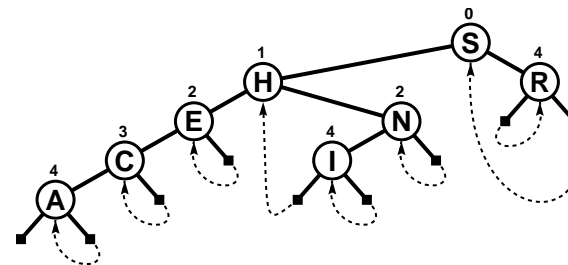


**INSERT:** search, then find leftmost bit that distinguishes new key from key found

Easy (external) case



Harder (internal) case



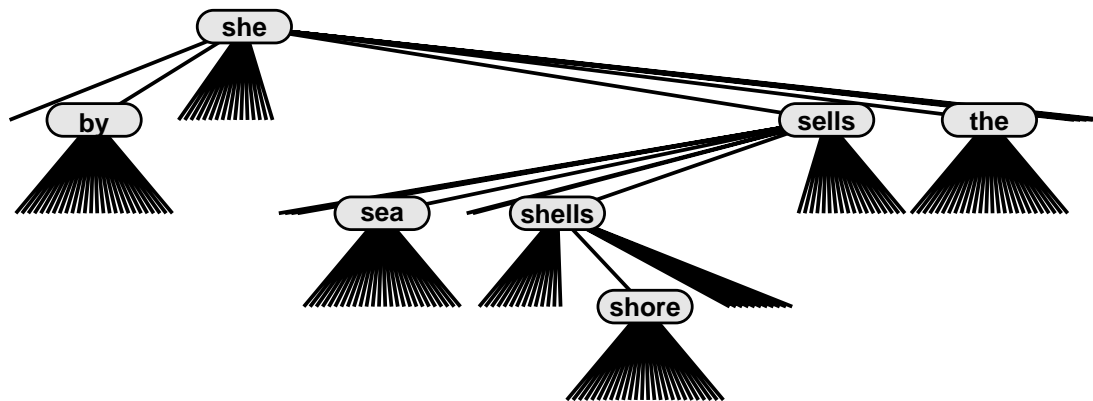
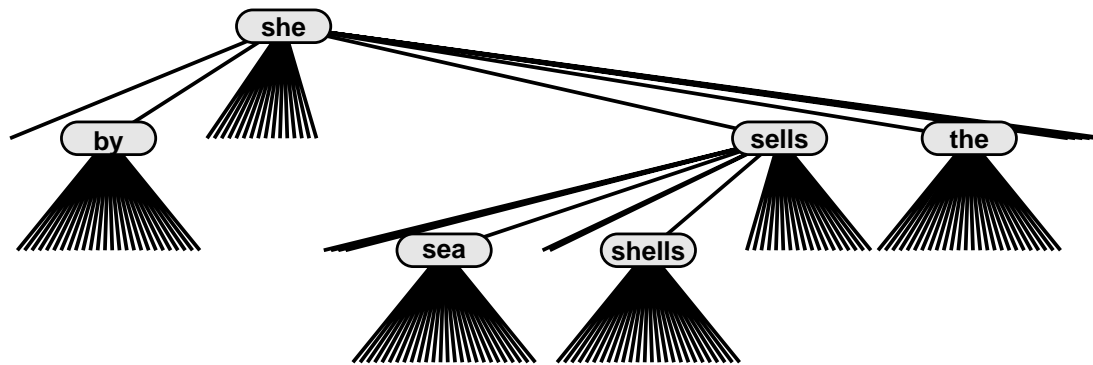
## R-way digital tree

Generalize DIGITAL TREE to R links per node  
R-way branching gives fast search

```
struct STnode
{ Item item; link next[26]; };
Item searchR(link h, Key v, int w)
{ int i = digit(v, w);
  if (h == z) return NULLitem;
  if eq(v, key(h->item)) return h->item;
  return searchR(h->next[i], v, w+1);
}
Item STsearch(Key v)
{ return searchR(head, v, 0); }
```

Simple, fast (but uses a lot of space)

# R-way digital tree example



## R-way digital tree analysis

N keys: N internal nodes, R links per node

**Space:**  $N \cdot R$

**Time:**  $\lg N / \lg R$  comparisons

**Ex:**  $R=26$ ,  $N=20000$

500,000 links

tree height 3-4

**Ex:**  $R=16$ ,  $N=1M$

16M links

tree height 5

**Plus:** one node type, easy implementation

**Minus:**

- full comparison at each node
- does not support SORT and SELECT

## R-way trie

Generalize TRIE to R-way branching

Nodes contain characters

R-way branching on next character

End-of-key options

- fixed-length keys
- prefix match
- prefix-free keys

end-of-key character

suffix trie

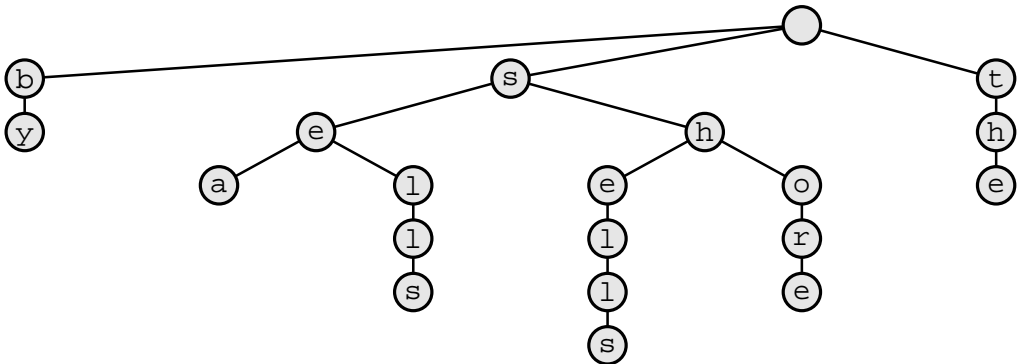
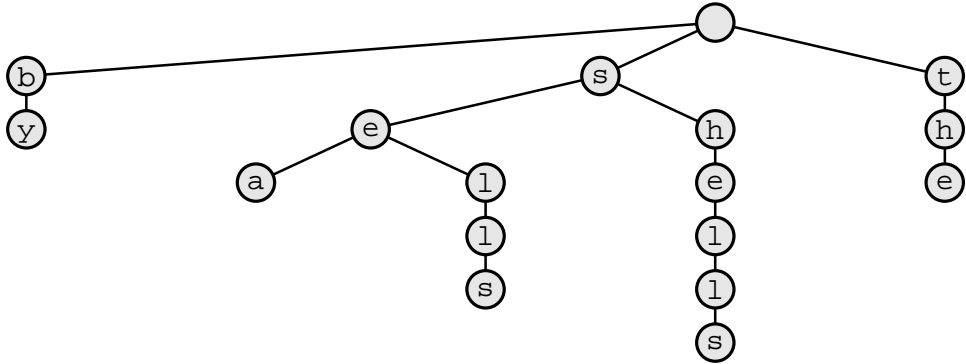
Maintain order in tree to support SORT and SELECT

## R-way trie search code

```
typedef struct STnode* link;
struct STnode
{ Item item; link next[R]; };
Item searchR(link h, Key v, int w)
{ int i = digit(v, w);
  if (h == z) return NULLitem;
  if (internal(h))
    return searchR(h->next[i], v, w+1);
  if eq(v, key(h->item)) return h->item;
  return NULLitem;
}
Item STsearch(Key v)
{ return searchR(head, v, 0); }
```



# R-way trie example (without null links)



## R-way trie existence table

### EXISTENCE TABLE

- no data in trie
- keys encoded in trie structure

Easy to implement, but may use excessive space

```
Item searchR(link h, Key v, int w)
{
    int i = digit(v, w);
    if (h == z) return NULLitem;
    if (i == NULLdigit) return v;
    return searchR(h->next[i], v, w+1);
}

Item STsearch(Key v)
{
    return searchR(head, v, 0);
}
```

Extend `digit(v, w)` implementation to

- return `NULLdigit` if `v` has fewer than `w` digits

## R-way trie insert code

```
link split(link p, link q, int w)
{ link t = NEW(NULLitem);
  int pd = digit(p->item, w),
      qd = digit(q->item, w);
  if (pd == qd)
      { t->next[pd] = split(p, q, w+1); }
  else { t->next[pd] = p; t->next[qd] = q; }
  return t;
}

link insertR(link h, Item item, int w)
{ Key v = key(item);
  int i = digit(v, w);
  if (h == z) return NEW(item);
  if (!internal(h))
      { return split(NEW(item), h, w); }
  h->next[i] = insertR(h->next[i], v, w+1);
  return h;
}

void STinsert(Item item)
{ head = insertR(head, item, 0); }
```

## R-way trie analysis

### Assumptions

- one-way links collapsed
- link to remainder of key in external nodes
- $N$  keys, total of  $C$  characters in keys
- approx.  $N$  internal nodes
- $R$  links per node

**Space:**  $N * R + C$

**Time:**  $\lg N / \lg R$  CHARACTER comparisons

**Ex:**  $R=26, N=20000$

520,000 links

tree height 3-4

**Ex:**  $R=16, N=1M$

16M links

tree height 5

## R-way trie summary

Faster than hashing

- successful search: no arithmetic
- unsuccessful search: don't need to examine whole key

Supports SORT, SELECT, other ADT operations

Problems

- eliminating 1-way branches
- multiple node types
- too much space for null links

## Correspondence with sorting algs

BSTs correspond to Quicksort recursive structure

Roles of TIME and SPACE interchanged

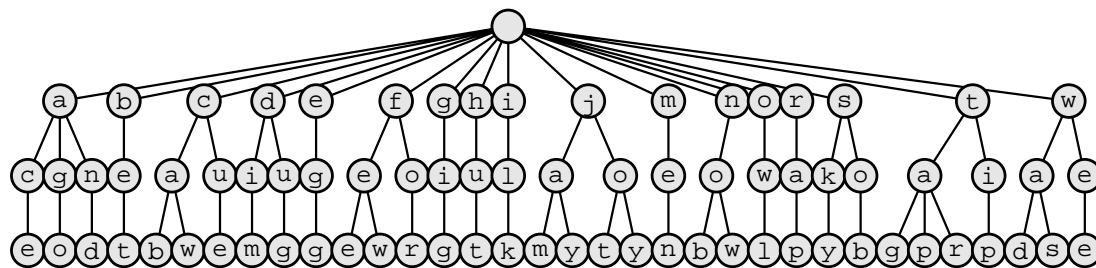
**TIME:**

- path in tree
- size of subfile in sort

**SPACE:**

- stack size in sort
- branching factor in tree

Tries correspond to MSD radix sort



3-way tries correspond to 3-way Quicksort

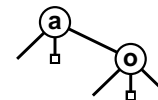
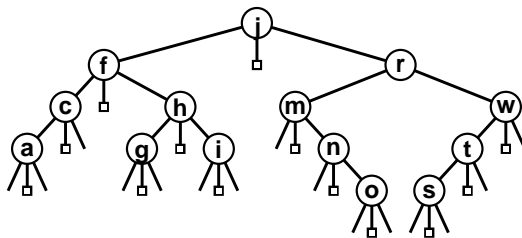
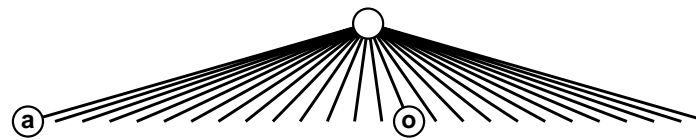
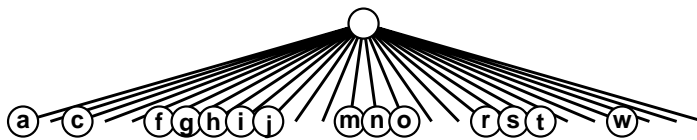
## Three-way radix tries

Nodes contain characters

three-way branching on next character

- left: key character less
- middle: key character equal
- right: key character greater

Equivalent to replacing trie node with BST on character



## Ternary search tries (TSTs)

Existence trie implementation OK

Adds factor of  $2 \ln M$  to search cost

- constant no. of BYTES for unsucc. search

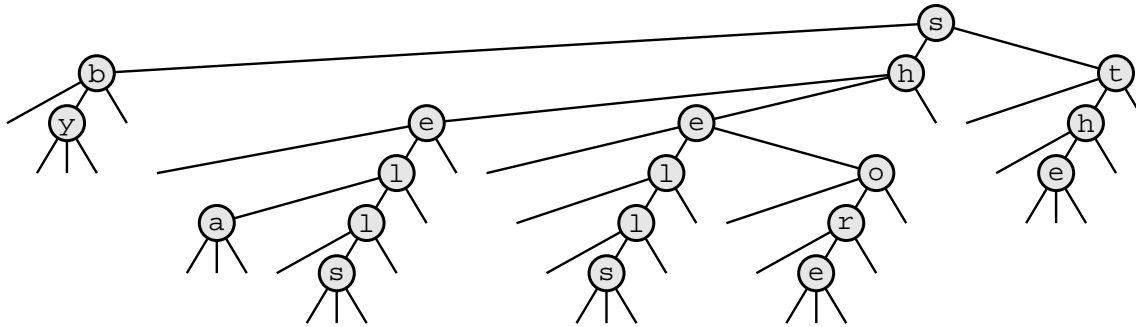
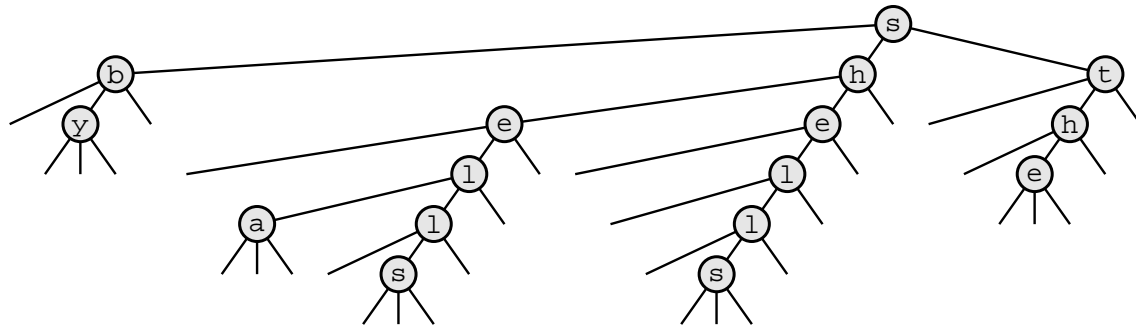
Easy recursive sort, selection

Most important advantages

- adapts well to strange keys
- uses just linear space
- supports full array of ADT operations

Faster than hashing, without wasting space

# TST example



## TST search implementation

### Code writes itself

```
Item searchR(link h, Key v, int w)
{
    int i = digit(v, w);
    if (h == z) return NULLitem;
    if (i == NULLdigit) return v;
    if (i < h->d) return searchR(h->l, v, w);
    if (i == h->d) return searchR(h->m, v, w+1);
    if (i > h->d) return searchR(h->r, v, w);
}

Item STsearch(char *v)
{
    return searchR(head, v, 0);
}
```

## TST insertion implementation

```
typedef struct STnode* link;
struct STnode
    { Item item; int d; link l, m, r; };
void STinit() { head = z; }
link NEW(int d)
    { link x = malloc(sizeof *x);
      x->d = d; x->l = z; x->m = z; x->r = z;
      return x;
    }
link insertR(link h, Item item, int w)
    { Key v = key(item);
      int i = digit(v, w);
      if (h == z) h = NEW(i);
      if (i == NULLdigit) return h;
      if (i < h->d) h->l = insertR(h->l, v, w);
      if (i == h->d) h->m = insertR(h->m, v, w+1);
      if (i > h->d) h->r = insertR(h->r, v, w);
      return h;
    }
void STinsert(Key key)
    { head = insertR(head, key, 0); }
```

## Empirical studies

random 32-bit keys

.	BUILD				SEARCH			
.	bst	dst	trie	pat	bst	dst	trie	pat
. 5000	4	5	7	7	3	2	3	2
. 12500	18	15	20	18	8	7	9	7
. 25000	40	36	44	41	20	17	20	17
. 50000	81	80	99	90	43	41	47	36
.100000	176	167	269	242	103	85	101	92
.200000	411	360	544	448	228	179	211	182

## Empirical studies (continued)

### words in Moby Dick

	BUILD				SEARCH			
	bst	hash	tst	fast	bst	hash	tst	fast
• 5000	5	4	3	3	3	2	2	1
• 12500	11	8	9	7	9	5	5	3
• 25000	23	15	17	13	19	12	10	7
• 50000	50	29	31	25	43	25	21	15

### library call numbers

	BUILD				SEARCH			
	bst	hash	tst	fast	bst	hash	tst	fast
• 5000	19	16	21	20	10	8	6	4
• 12500	48	48	54	97	29	27	15	14
• 25000	118	99	188	156	67	59	36	30
• 50000	230	191	333	255	137	113	70	65

fast: TST with R\*R-way branching at the root