COS 226 Lecture 10: Radix trees and tries

Symbol Table, Dictionary

- records with keys
- INSERT
- SEARCH

Balanced trees, randomized trees

- use O(lgN) comparisons

Hashing

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

Is key length required?

- (no)

Are comparisons necessary?

- (no)

Best possible: examine lgN BITS

Digital trees

Easy way to balance tree

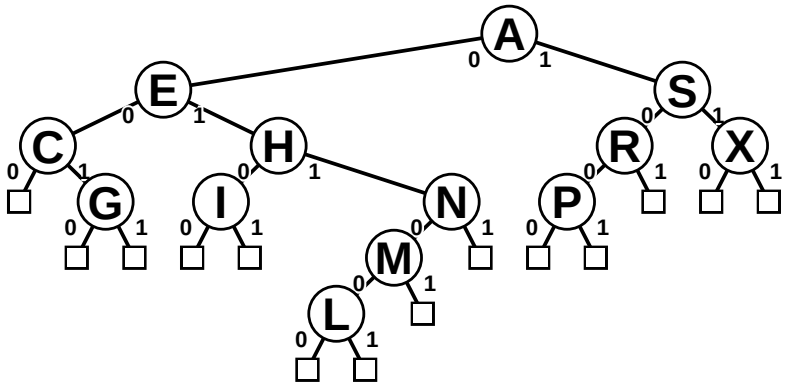
Use bits of key to direct search

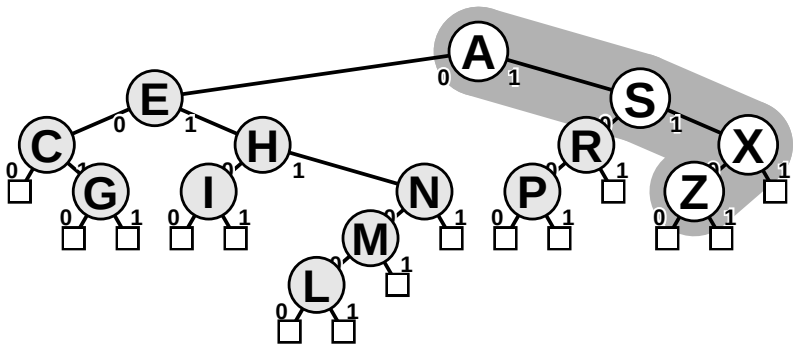
Otherwise identical to BST

```
recType DSTsearch(ST head, keyType v, int b)
   {
      if (head == z) return nullrecord;
      if (v == key(head->rec)) return head->rec;
      if (!digit(v, b))
         return DSTsearch(head->l, v, b+1);
      else
         return DSTsearch(head->r, v, b+1);
   }
ST DSTinsert(ST head, recType rec, int b)
   { keyType v = key(rec);
      if (head == z)
         return head = NEW(rec, z, z);
      if (!digit(v, b))
         head->l = DSTinsert(head->l, rec, b+1);
      else
         head->r = DSTinsert(head->r, rec, b+1);
      (head->N)++;
      return head;
   }
```

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



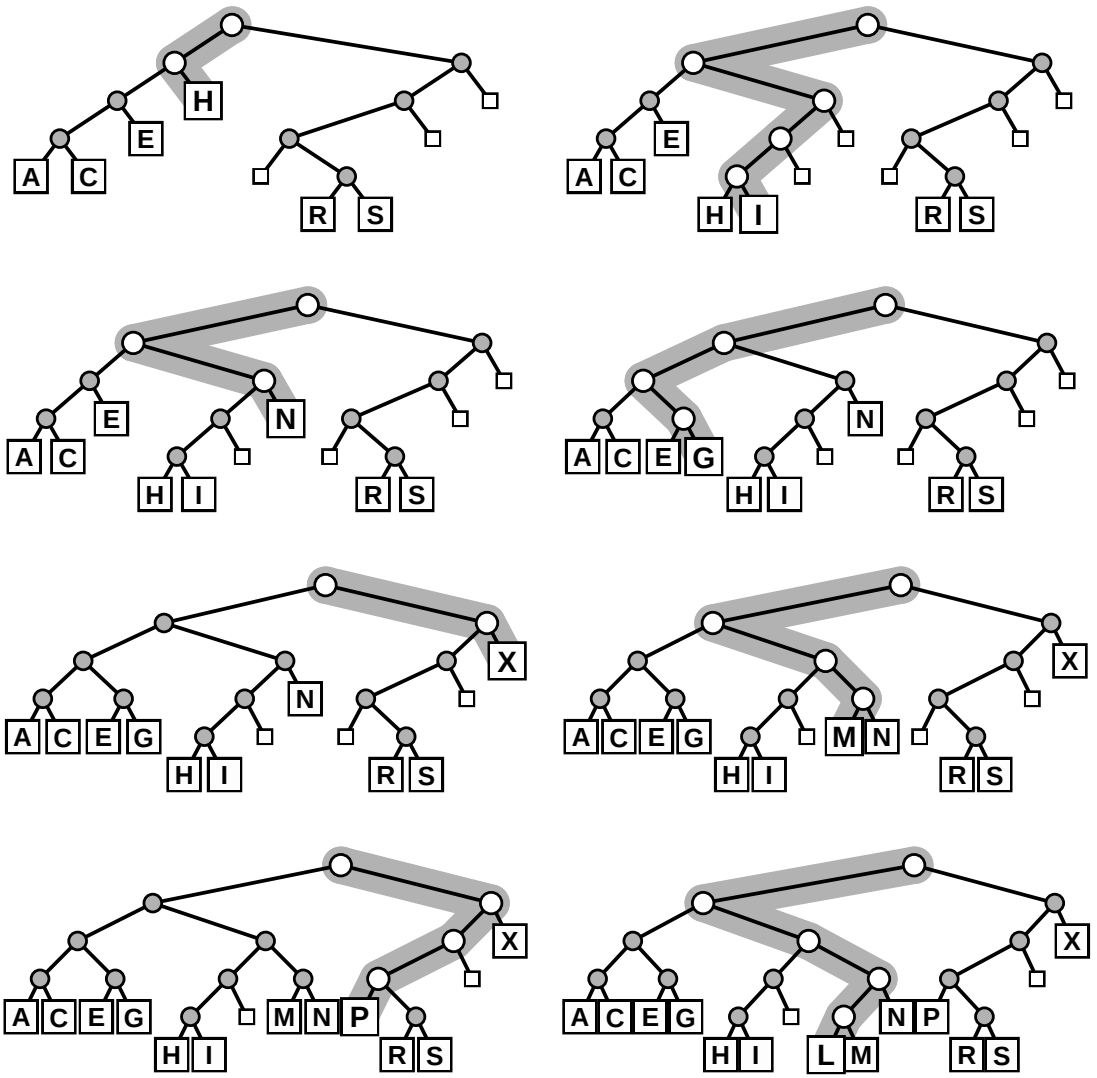


Tries

Branch according to bits in keys

No keys in internal nodes

Records/keys in external nodes



Structure depends on keys, not insertion order

Examine lgN BITS to distinguish key

independent of key length!

Problems:

44% space waste from 1-way branching

multiple node types

Patricia tries

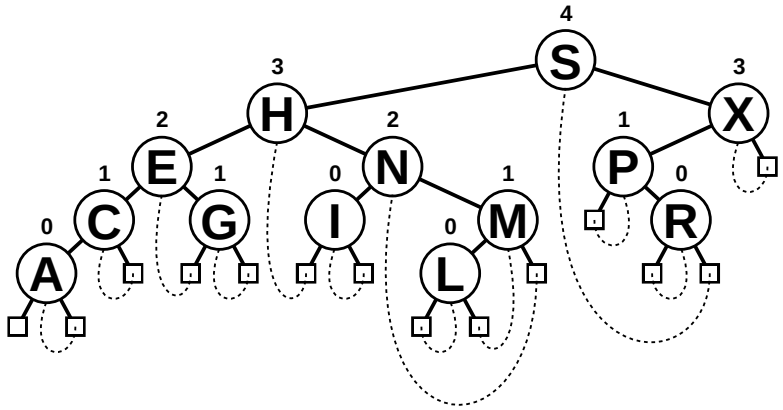
Digression: cute nomenclature

Practical
Algorithm
To
Retrieve
Information
Coded
In
Alphanumeric

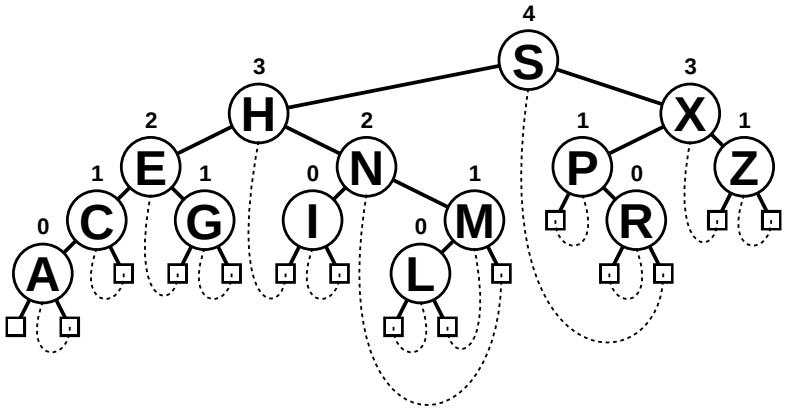
information reTRIEval
(but pronounced "try")

Patricia:
* collapse one-way branches in tries
* "thread" tree to eliminate multiple node types
Quintessential search algorithm

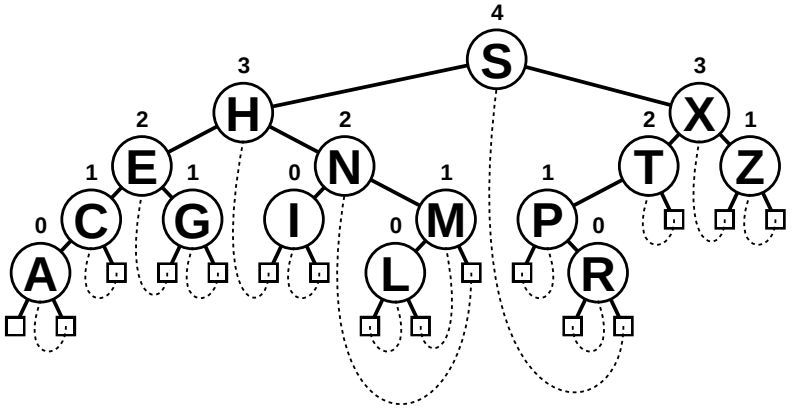
Patricia tree insertion



Easy (external) case



Harder (internal) case



M–way digital tree

Generalize to M links per node

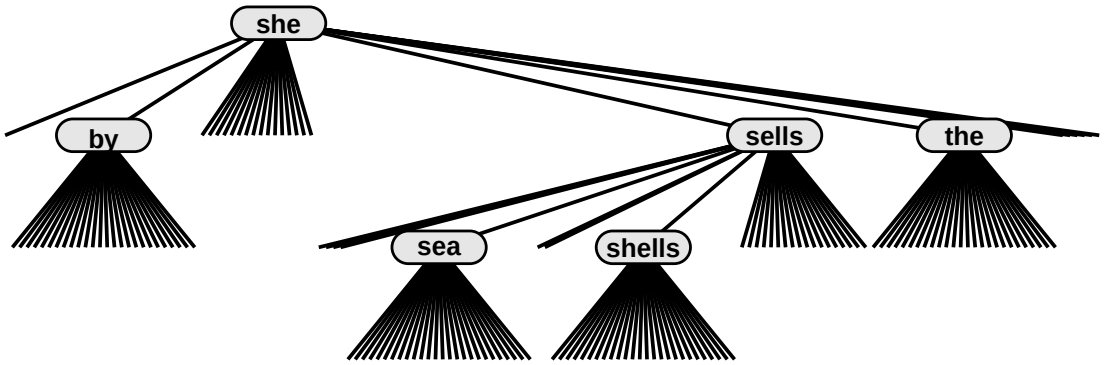
M–way branching gives fast search

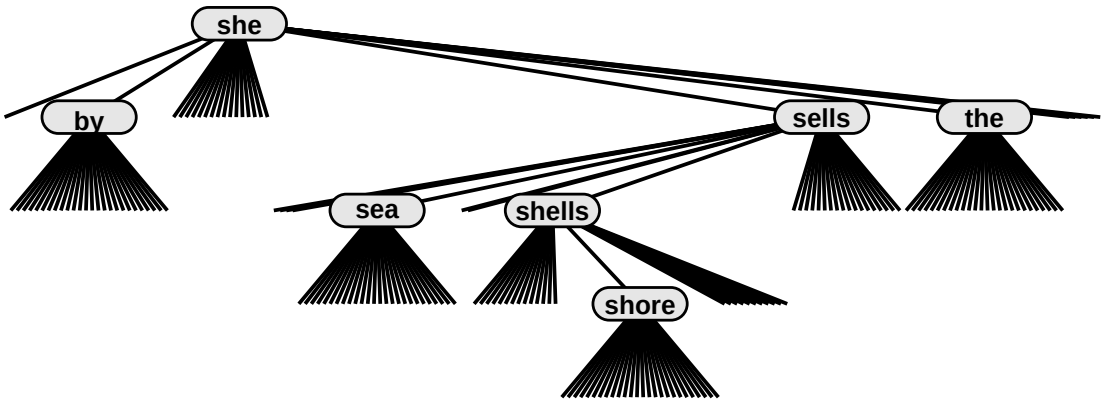
```
struct DSTnode
   { recType rec; DSTptr next[26]; };
DSTptr NEW(recType);

recType DSTsearchR(DSTptr x, char *v, int j)
   { int i = v[j] - 'a';
      if (x == z) return nullRecord;
      if (eq(v, key(x->rec))) return x->rec;
      return DSTsearchR(x->next[i], v, j+1);
   }

recType DSTsearch(char *v)
   { return DSTsearchR(head, v, 0); }
```

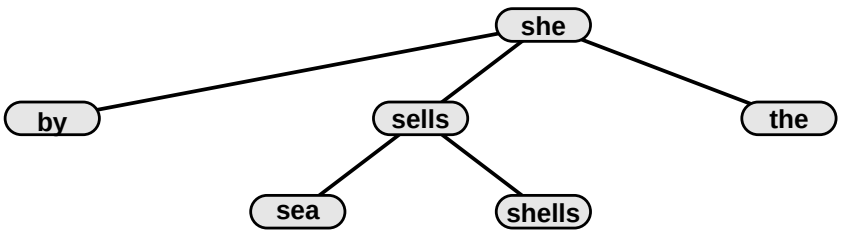
Simple, fast

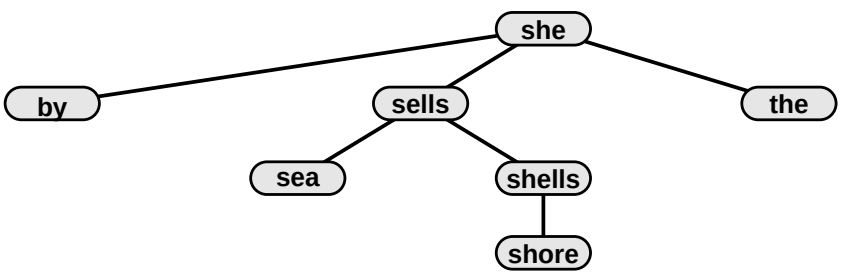
(but uses a lot of space)

M-way digital tree example









M-way digital tree analysis

N keys

N internal nodes
M links per node

Space: N*M
Time: lgN/lgM comparisons

Ex: M=26, N=20000
500,000 links
tree height 3-4

Ex: M=16, N=1M
16M links
tree height 5

Advantage: one node type, easy implementation
Disadvantage: full comparison at each node

M-way digital tree insertion code

```
void DSTinit()
   { head = z;}
DSTptr NEW(recType rec)
   { int i;
      DSTptr x = (DSTptr) malloc(sizeof *x);
      x->rec = rec;
      for (i = 0; i < 26; i++) x->next[i] = z;
      return x;
   }
DSTptr DSTinsertR(DSTptr x, recType rec, char *v)
   { int i;
      if (x == z) return NEW(rec);
      if (v[0] == ' ') return x;
      i = v[0] - 'a';
      x->next[i] = DSTinsertR(x->next[i], rec, v+1)
      return x;
   }
void DSTinsert(recType rec)
   { head = DSTinsertR(head, rec, key(rec)); }
```

M-way trie

Generalize trie to M-way branching

Nodes contain characters

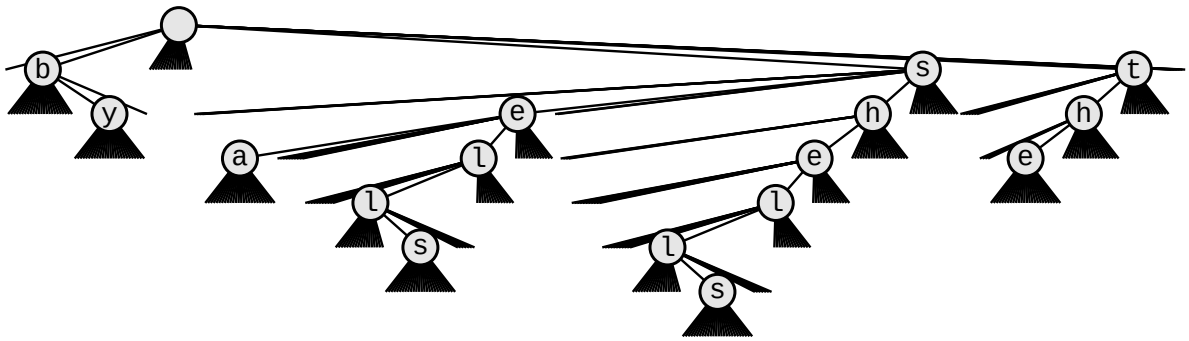
M-way branching on next character

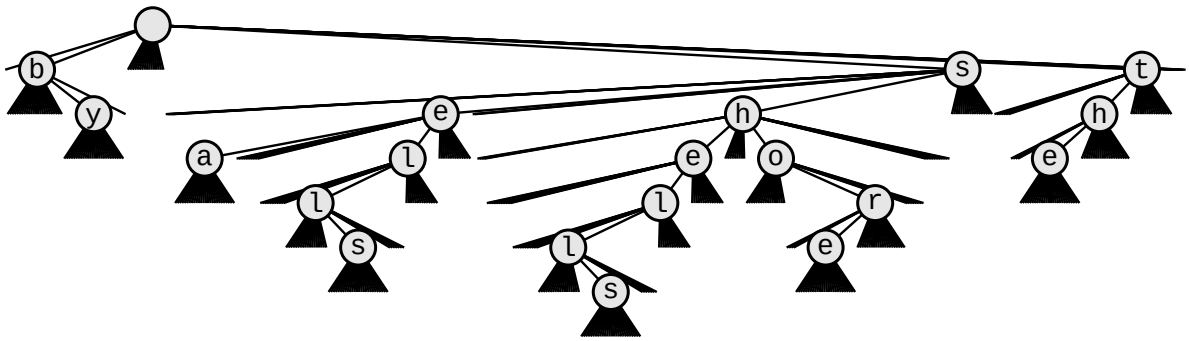
```
typedef struct TRIEnode* TRIEptr;
struct TRIEnode
   { char ch; TRIEptr next[26]; };

int TRIEsearchR(TRIEptr x, char *v)
   { int i = *v - 'a';
      if (x == z) return 0;
      if (*v == ' ') return 1;
      return TRIEsearchR(x->next[i], v+1);
   }
int TRIEsearch(char *v)
   { return TRIEsearchR(head, v); }
```

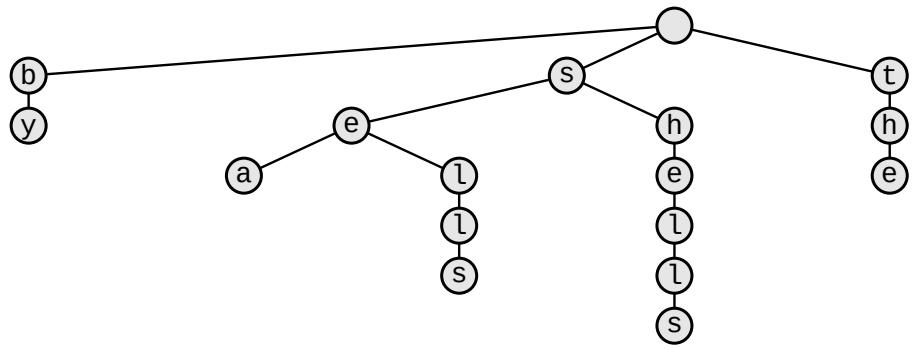
Return match on prefix, or make keys prefix-free

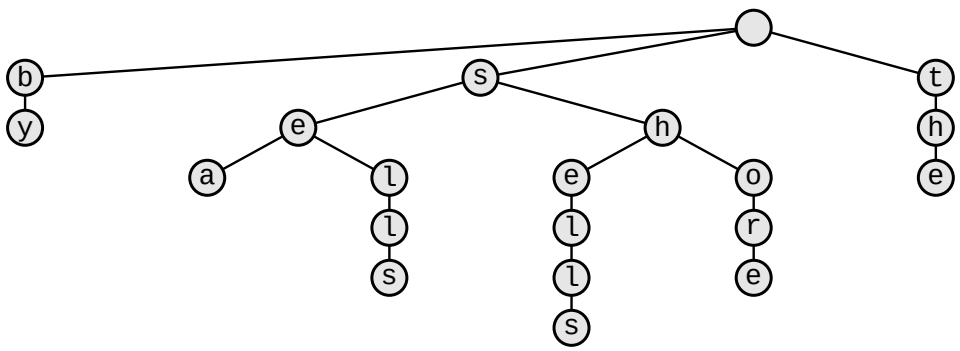
- end-of-key character
- suffix trie

M-way trie example









Insertion in M-way trie

```
void TRIEinit()
   { head = z;}
TRIEptr NEW(char ch)
   { int i;
      TRIEptr x = (TRIEptr) malloc(sizeof *x);
      x->ch = ch;
      for (i = 0; i < 26; i++) x->next[i] = z;
      return x;
   }
TRIEptr TRIEinsertR(TRIEptr x, char *v)
   { int i;
      if (x == z) x = NEW(' ');
      if (*v == ' ') return x;
      i = *v - 'a';
      x->next[i] = TRIEinsertR(x->next[i], v+1);
      x->next[i]->ch = *v;
      return x;
   }
void TRIEinsert(keyType key)
   { head = TRIEinsertR(head, key); }
```

M-way trie analysis

Assume

- one-way links collapsed
- chars in internal nodes
- link to remainder of key in external nodes

N keys, total of C characters in keys
approx. N internal nodes
M links per node

Space: N*M + C
Time: lgN/lgM CHARACTER comparisons

Ex: M=26, N=20000
- 500,000 links
- tree height 3-4

Ex: M=16, N=1M
- 16M links
- tree height 5

Faster than hashing
- particularly for unsuccessful search

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

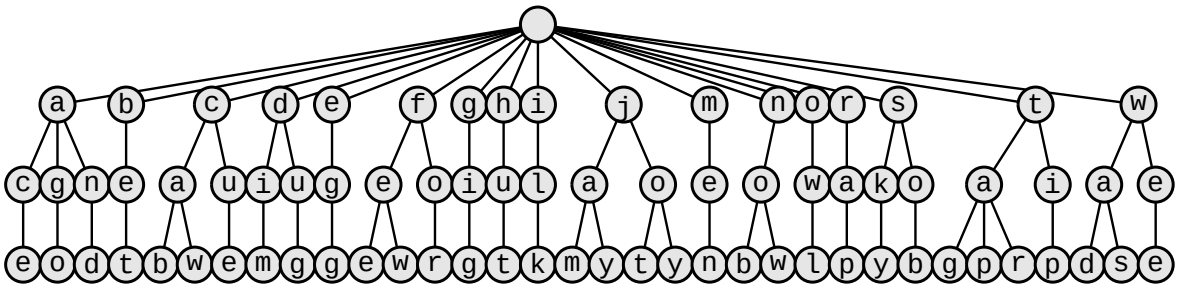
Correspondence with sorting algs

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

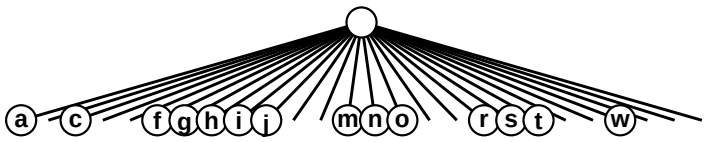
Nodes contain characters

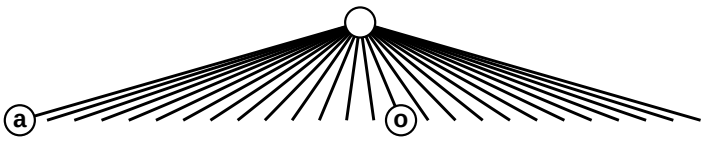
three-way branching on next character

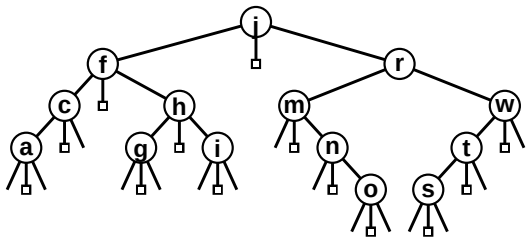
- left: key character less
- middle: key character equal
- right: key character greater

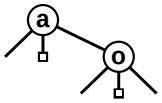
Equivalent to replacing trie node with

- BST on character









Adds factor of 2lnM to search cost

- about 1.44 lgN BYTES for unsucc. search

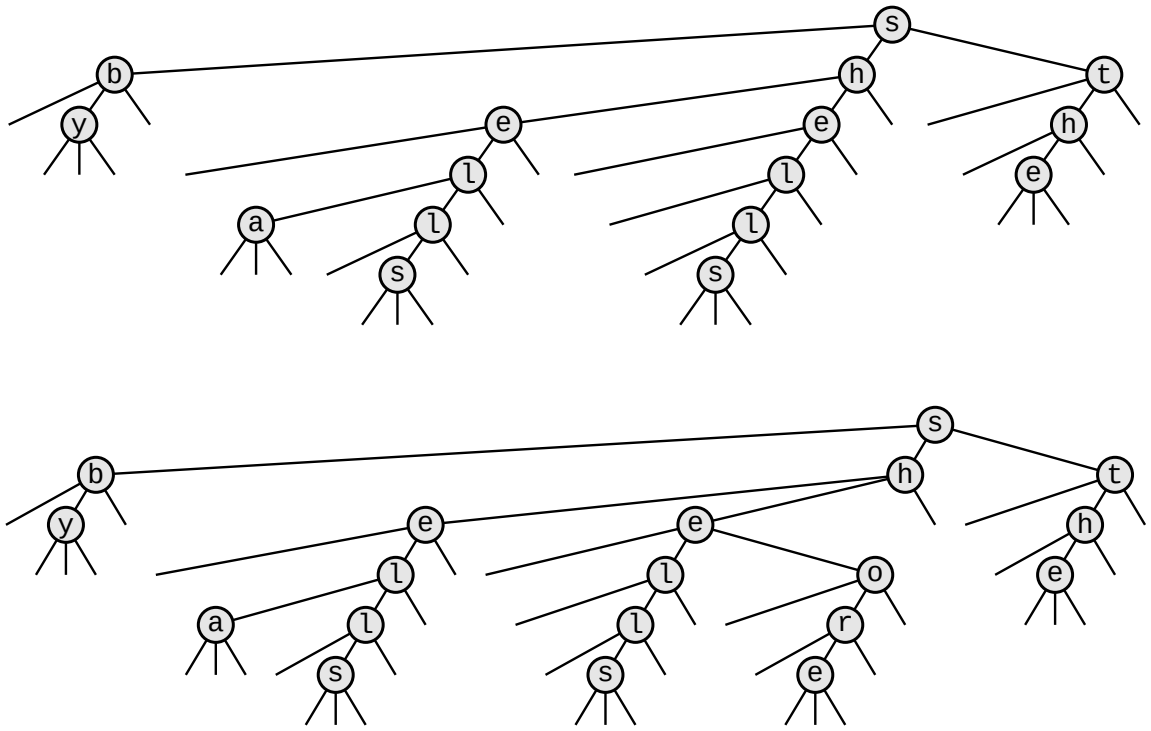
Easy recursive sort

Most important advantage:

- adapts well to strange keys

Faster than hashing

Three-way radix search trie example




```
int RSTsearchR(RSTptr x, char *v)
   {
      if (*v == ' ') return 1;
      if (x == z) return 0;
      if (*v < x->ch)
         return RSTsearchR(x->l, v);
      if (*v == x->ch)
         return RSTsearchR(x->m, v+1);
      if (*v > x->ch)
         return RSTsearchR(x->r, v);
   }
int RSTsearch(char *v)
   { return RSTsearchR(head, v); }
```

Insertion in three-way radix trie

```
typedef struct RSTnode* RSTptr;
struct RSTnode
   { char ch; RSTptr l, m, r; };
RSTptr NEW(char ch)
   { int i;
      RSTptr x = (RSTptr) malloc(sizeof *x);
      x->ch = ch; x->l = z; x->m = z; x->r = z;
      return x;
   }
RSTptr RSTinsertR(RSTptr x, char *v)
   {
      if (x == z) x = NEW(*v);
      if (*v == ' ') return z;
      if (*v < x->ch)
         x->l = RSTinsertR(x->l, v);
      if (*v == x->ch)
         x->m = RSTinsertR(x->m, v+1);
      if (*v > x->ch)
         x->r = RSTinsertR(x->r, v);
      return x;
   }
void RSTinsert(keyType key)
   { head = RSTinsertR(head, key); }
void RSTinit()
   { head = z;}
```