Hash Collisions

❖ Hashing: Reminder

Goal is to use keys as indexes, e.g.

```
courses["COMP3311"] = "Database Systems";
printf("%s\n", courses["COMP3311"]);
```

Since strings can't be indexes in C, use via a hash function, e.g.

```
courses[h("COMP3311")] = "Database Systems";
printf("%s\n", courses[h("COMP3311")]);
```

Hash function `h` converts key → integer and uses that as the index.

Problem: collisions, where k ≠ j but hash(k,N) = hash(j,N)

❖ Collision Resolution

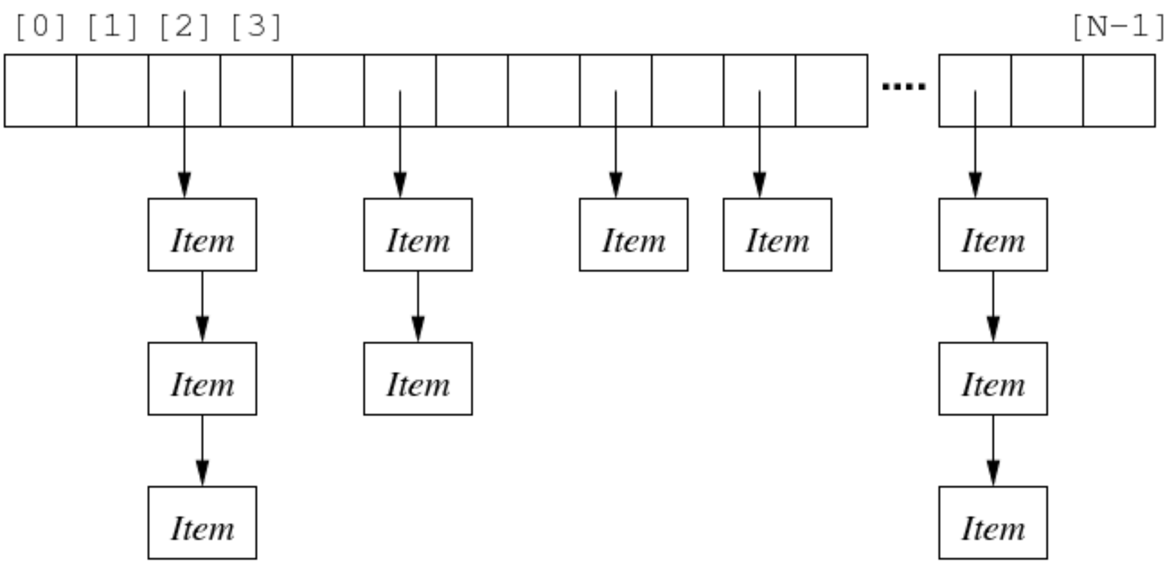
Three approaches to dealing with hash collisions:

- allow multiple `Items` at a single array location
  - e.g. array of linked lists (but worst case is *O(N)*)
- systematically compute new indexes until find a free slot
  - need strategies for computing new indexes (aka probing)
- increase the size of the array
  - needs a method to "adjust" `hash()` (e.g. linear hashing)

❖ Separate Chaining

Solve collisions by having multiple items per array entry.

Make each element the start of linked-list of Items.



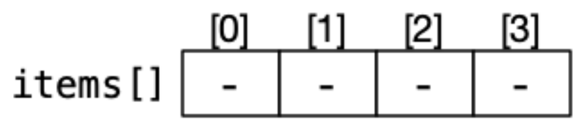
All items in a given list have the same **`hash()`** value

❖ ... Separate Chaining

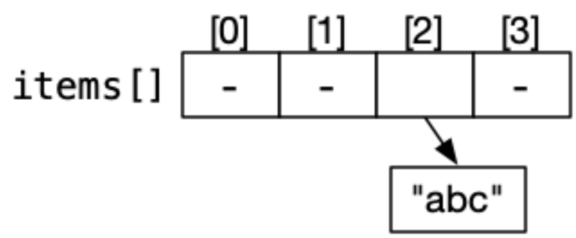
Example of separate chaining ...

h("abc") = 2, h("def") = 1, h("ghi") = 0, h("jkl") = 2, h("mno") = 1

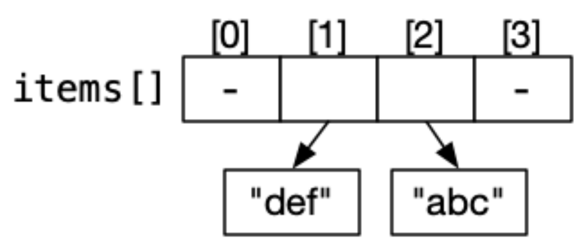
Initially



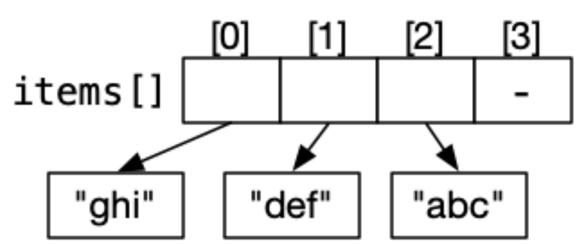
After inserting "abc" (h=2)



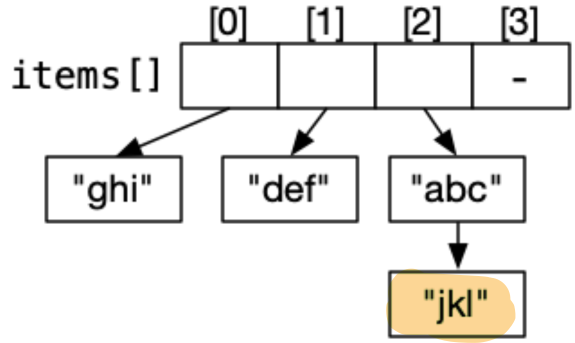
After inserting "def" (h=1)



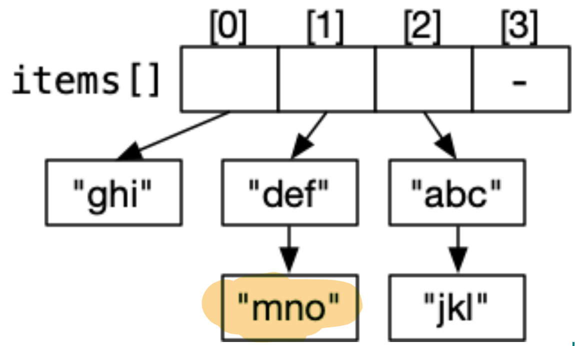
After inserting "ghi" (h=0)



After inserting "jkl" (h=2)



After inserting "mno" (h=1)



❖ ... Separate Chaining

Concrete data structure for hashing via chaining

```
typedef struct HashTabRep {
   List *lists; // array of Lists of Items
   int  N;      // # elements in array
   int  nitems; // # items stored in HashTable
} HashTabRep;

HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->lists = malloc(N*sizeof(List));
   assert(new->lists != NULL);
   for (int i = 0; i < N; i++)
      new->lists[i] = newList();
   new->N = N; new->nitems = 0;
   return new;
}
```

❖ ... Separate Chaining

Using the **`List`** ADT, search becomes:

```
#include "List.h"
Item *HashGet(HashTable ht, Key k)
{
   int i = hash(k, ht->N);
   return ListSearch(ht->lists[i], k);
}
```

Even without **`List`** abstraction, easy to implement.

Using sorted lists gives only small performance gain.

❖ ... Separate Chaining

Other list operations are also simple:

```
#include "List.h"

void HashInsert(HashTable ht, Item it) {
   Key k = key(it);
   int i = hash(k, ht->N);
   ListInsert(ht->lists[i], it);
}
void HashDelete(HashTable ht, Key k) {
   int i = hash(k, ht->N);
   ListDelete(ht->lists[i], k);
}
```

insert

Essentially: select a list; operate on that list.

❖ ... Separate Chaining

Cost analysis:

- N array entries (slots), M stored items
- average list length $L = M/N$
- best case: all lists are same length L
- worst case: one list of length M ($h(k)=0$) array[0]全有
- searching within a list of length n:
    - best: 1, worst: n, average: $n/2$ ⇒ $O(n)$
- M small
- if good hash and $M \le N$, cost is 1
- if good hash and $M > N$, cost is $(M/N)/2$

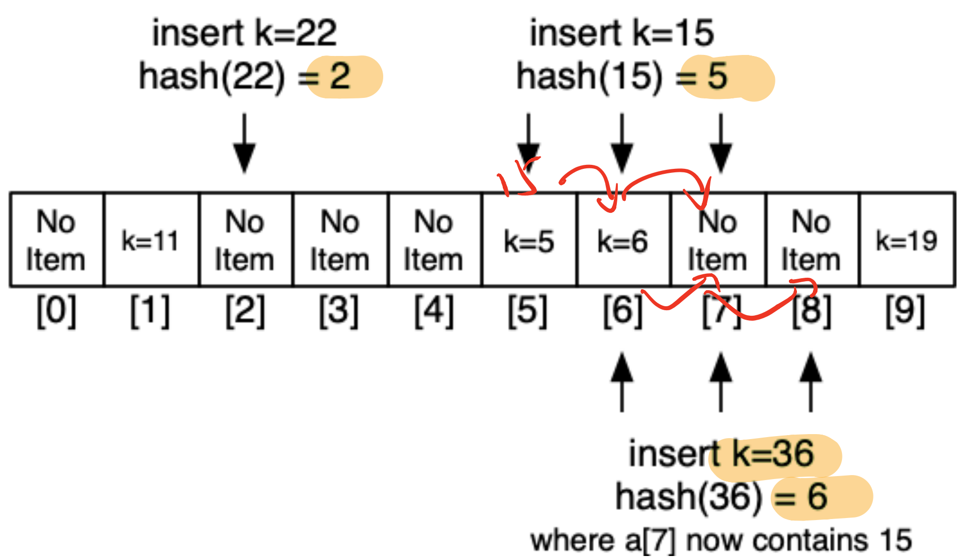
Ratio of items/slots is called **load α = M/N**

❖ Linear Probing

Collision resolution by finding a new location for `Item`

- hash indicates slot *i* which is already used
- try next slot, then next, until we find a free slot
- insert item into available slot

Examples:



❖ ... Linear Probing

Concrete data structures for hashing via linear probing:

```
typedef struct HashTabRep {
   Item **items; // array of pointers to Items
   int  N;       // # elements in array
   int  nitems;  // # items stored in HashTable
} HashTabRep;

HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->items = malloc(N*sizeof(Item *));
   assert(new->items != NULL);
   for (int i = 0; i < N; i++) new->items[i] = NULL;
   new->N = N; new->nitems = 0;
   return new;
}
```

❖ ... Linear Probing

Insert function for linear probing:

eg. (12, "asas")
(key, value)

```
void HashInsert(HashTable ht, Item it)
{
    assert(ht->nitems < ht->N);
    int N = ht->N;
    Key k = key(it);
    Item **a = ht->items;
    int i = hash(k,N);
    for (int j = 0; j < N; j++) {
        if (a[i] == NULL) break;
        if (equal(k,key(*(a[i])))) break;
        i = (i+1) % N;
    }
    if (a[i] == NULL) ht->nitems++;
    if (a[i] != NULL) free(a[i]);
    a[i] = copy(it);
}
```

array[]

已经有 it了

else

→ 已经有 it, duplicate

put it into a[i]

❖ ... Linear Probing

Search function for linear probing:

```
Item *HashGet(HashTable ht, Key k)
{
   int N = ht->N;
   Item **a = ht->items;
   int i = hash(k,N);
   for (int j = 0; j < N; j++) {  没有
      if (a[i] == NULL) break;
      if (equal(k,key(*(a[i]))))  → found
         return a[i];
      i = (i+1) % N;
   }
   return NULL;
}
```

❖ ... Linear Probing

Search cost analysis:

- cost to reach first **`Item`** is *O(1)*
- subsequent cost depends how much we need to scan
- affected by **load** α = M/N (i.e. how "full" is the table)
- average cost for successful search = *0.5*(1 + 1/(1-α))*
- average cost for unsuccessful search = *0.5*(1 + 1/(1-α)²)*

half full
nearly full

Example costs (assuming large table, e.g. *N>100*):

load (α)	0.50	0.67	0.75	0.90
search hit	1.5	2.0	3.0	5.5
search miss	2.5	5.0	8.5	55.5

expensive.

Assumes reasonably uniform data and good hash function.

❖ ... Linear Probing

Deletion slightly tricky for linear probing.

Need to ensure no **`NULL`** in middle of "probe path"
(i.e. previously relocated items moved to appropriate location)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	k=11	No Item	No Item	No Item	k=5	k=6	k=15	k=25	k=19

delete(k=5) → ...

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	k=11	No Item	No Item	No Item	k=15	k=6	k=25	No Item	k=19

❖ ... Linear Probing

Delete function for linear probing:

```
void HashDelete(HashTable ht, Key k)
{
   int N = ht->N;
   Item *a = ht->items;
   int i = hash(k,N);
delet   for (int j = 0; j < N; j++) {
      if (a[i] == NULL) return; // k not in table
      if (equal(k,key(*(a[i])))) break;
      i = (i+1) % N;
   }
   free(a[i]); a[i] = NULL; ht->nitems--;
   // clean up probe path
   i = (i+1) % N; i++
   while (a[i] != NULL) {
remove      Item it = *(a[i]);
      a[i] = NULL; // remove 'it'
      ht->nitems--;
      HashInsert(ht, it); // insert 'it' again
      i = (i+1) % N; i++
   }
}
```

h("ab") = 2, h("cd") = 1, h("ef") = 0, h("gh") = 2, h("ij") = 1

		[0]	[1]	[2]	[3]	[4]	[5]
Initially	`items[]`	-	-	-	-	-	-
After inserting "ab" (h=2)	`items[]`	-	-	"ab"	-	-	-
After inserting "cd" (h=1)	`items[]`	-	"cd"	"ab"	-	-	-
After inserting "ef" (h=0)	`items[]`	"ef"	"cd"	"ab"	-	-	-
After inserting "gh" (h=2)	`items[]`	"ef"	"cd"	"ab"	"gh"	-	-
After inserting "ij" (h=1)	`items[]`	"ef"	"cd"	"ab"	"gh"	"ij"	-

"ab" ✓

		[0]	[1]	[2]	[3]	[4]	[5]
After deleting "ab" (h=2)	`items[]`	"ef"	"cd"	"gh"	"ij"	-	-
After deleting "cd" (h=1)	`items[]`	"ef"	"ij"	"gh"	-	-	-

❖ ... Linear Probing

A problem with linear probing: clusters

E.g. insert 5, 6, 15, 16, 14, 25, with hash(k) = k%10

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-	1	-	-	4	-	-	-	-	-
-	1	-	-	4	5	-	-	-	-
-	1	-	-	4	5	6	-	-	-

15%10=5

-	1	-	-	4	5	6	15	-	-

16%10=6

-	1	-	-	4	5	6	15	16	-

14%10=4

-	1	-	-	4	5	6	15	16	14
25	1	-	-	4	5	6	15	16	14

25%10=5

❖ Double Hashing

Double hashing improves on linear probing:

- by using an increment which ...
  - is based on a secondary hash of the key
  - ensures that all elements are visited
    (can be ensured by using an increment which is relatively prime to N)
- tends to eliminate clusters ⇒ shorter probe paths

To generate relatively prime

- set table size to prime e.g. N=127
- **`hash2()`** in range [1..N1] where N1 < 127 and prime

❖ ... Double Hashing

Concrete data structures for hashing via double hashing:

```
typedef struct HashTabRep {
   Item **items; // array of pointers to Items
   int  N;       // # elements in array
   int  nitems;  // # items stored in HashTable
   int  nhash2;  // second hash mod
} HashTabRep;

#define hash2(k,N2) (((k)%N2)+1)

HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->items = malloc(N*sizeof(Item *));
   assert(new->items != NULL);
   for (int i = 0; i < N; i++)
      new->items[i] = NULL;
   new->N = N; new->nitems = 0;
   new->nhash2 = findSuitablePrime(N);
   return new;
}
```

❖ ... Double Hashing

Search function for double hashing:

```
Item *HashGet(HashTable ht, Key k)
{
   Item **a = ht->items;
   int N = ht->N;
   int i = hash(k,N); N%k
   int incr = hash2(k,ht->nhash2);
   for (int j = 0, j < N; j++) {
      if (a[i] == NULL) break;  // k not found
      if (equal(k,key(*(a[i]))) return a[i];
      i = (i+incr) % N;
   }
   return NULL;
}
```

❖ ... Double Hashing

Insert function for double hashing:

```
void HashInsert(HashTable ht, Item it)
{
   assert(ht->nitems < ht->N); // table full
   Item **a = ht->items;
   Key k = key(it);
   int N = ht->N;
   int i = hash(k,N);
   int incr = hash2(k,ht->nhash2);
   for (int j = 0, j < N; j++) {
      if (a[i] == NULL) break;
      if (equal(k,key(*(a[i])))) break;
      i = (i+incr) % N;
   }
   if (a[i] == NULL) ht->nitems++;
   if (a[i] != NULL) free(a[i]);
   a[i] = copy(it);
}
```

❖ ... Double Hashing

Search cost analysis:

- cost to reach first `Item` is *O(1)*
- subsequent cost depends how much we need to scan
- affected by load α = M/N (i.e. how "full" is the table)
- average cost for successful search = $\frac{1}{\alpha} ln(\frac{1}{1-\alpha})$
- average cost for unsuccessful search = $\frac{1}{1-\alpha}$

Costs for double hashing (assuming large table, e.g. *N>100*):

load (α)	0.5	0.67	0.75	0.90
search hit	1.4	1.6	1.8	2.6
search miss	1.5	2.0	3.0	5.5

Can be significantly better than linear probing

- especially if table is heavily loaded

❖ Hashing Summary

Collision resolution approaches:

M>M

- chaining: easy to implement, allows α > 1
- linear probing: fast if α ≪ 1, complex deletion
- double hashing: faster than linear probing, esp for α ≅ 1

Only chaining allows α > 1, but performance poor when α ≫ 1

For arrays, once *M* exceeds initial choice of *N*,

- need to expand size of array (*N*)
- problem: hash function relies on *N*,
  so changing array size potentially requires rebuiling whole table
- dynamic hashing methods exist to avoid this

Tries

❖ Tries

A trie ...

- is a data structure for representing a set of strings
  - e.g. all the distinct words in a document, a dictionary etc.
- supports string matching queries in *O(L)* time
  - *L* is the length of the string being searched for

Note: generally assume "string" = character string; could be bit-string

Note: Trie comes from *retrieval* ; but pronounced as "try" not "tree"

❖ ... Tries

Each node in a trie ...

- contains one part of a key (typically one character)
- may have up to 26 children [illegible]
- may be tagged as a "finishing" node
- but even "finishing" nodes may have children
- may contain other data for application (e.g. word frequency)

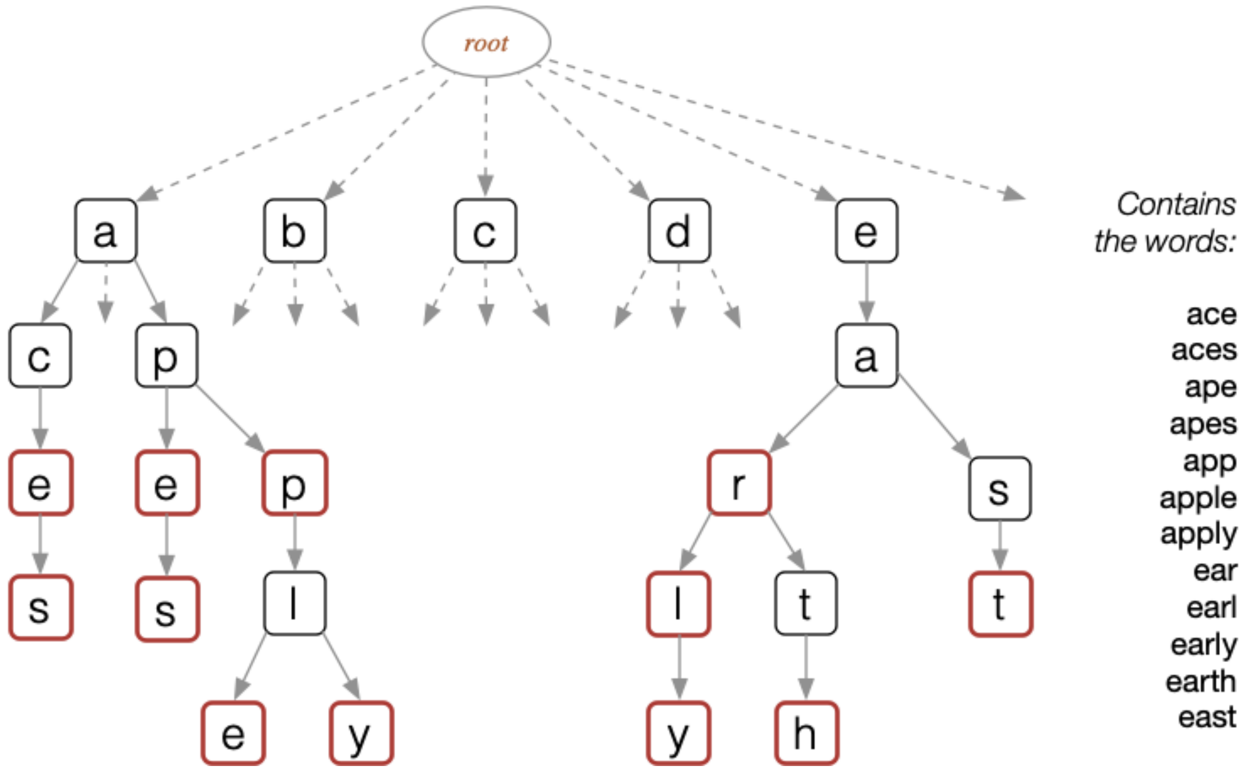
A "finishing" node marks the end of one key

- this key may be a prefix of another key stored in trie

Depth d of trie = length of longest key value

❖ ... Tries

Trie example:



❖ ... Tries

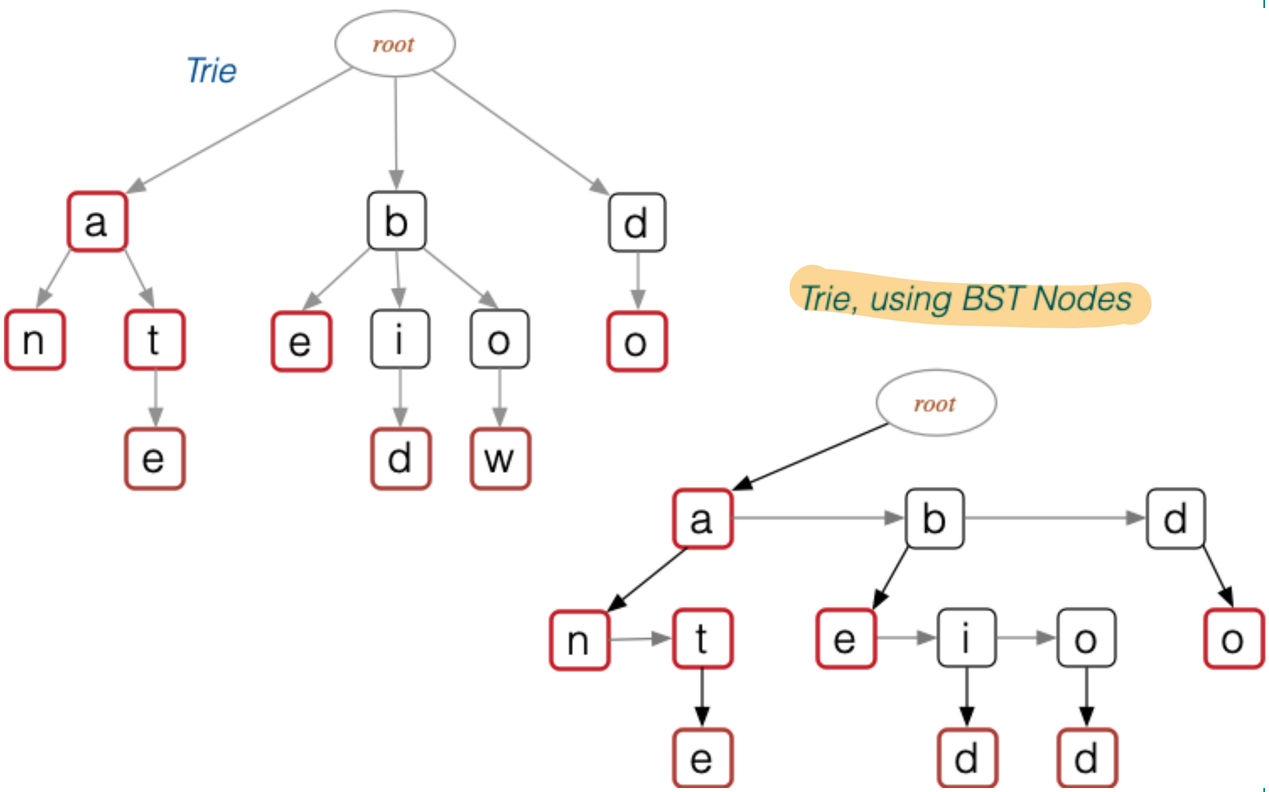
Possible trie representation:

```
#define ALPHABET_SIZE 26

typedef struct Node *Trie;

typedef struct Node {
   char onechar;          // current char in key
   Trie child[ALPHABET_SIZE];
   bool finish;           // last char in key?
   Item data;             // no Item if !finish
} Node;

typedef char *Key;     // just lower-case letters
```

"we"

❖ ... Tries

Note: Can also use BST-like nodes (cf. red-black trees) ...

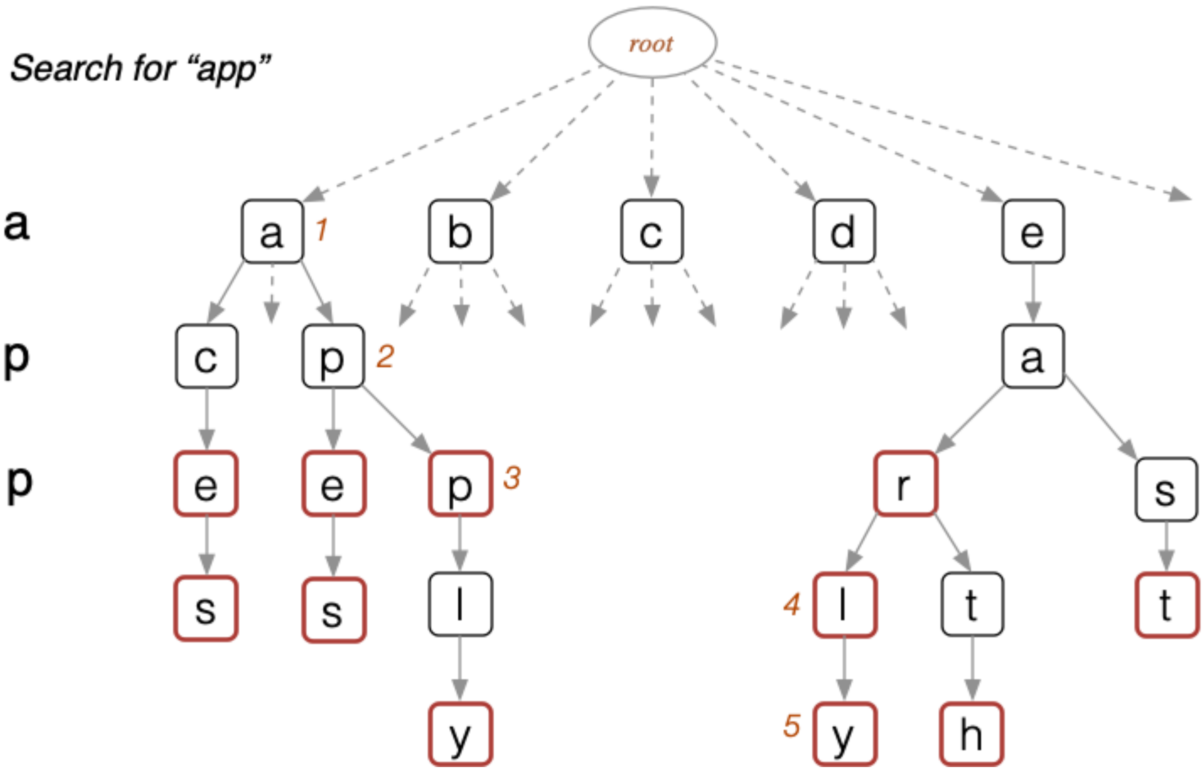


❖ Searching in Tries

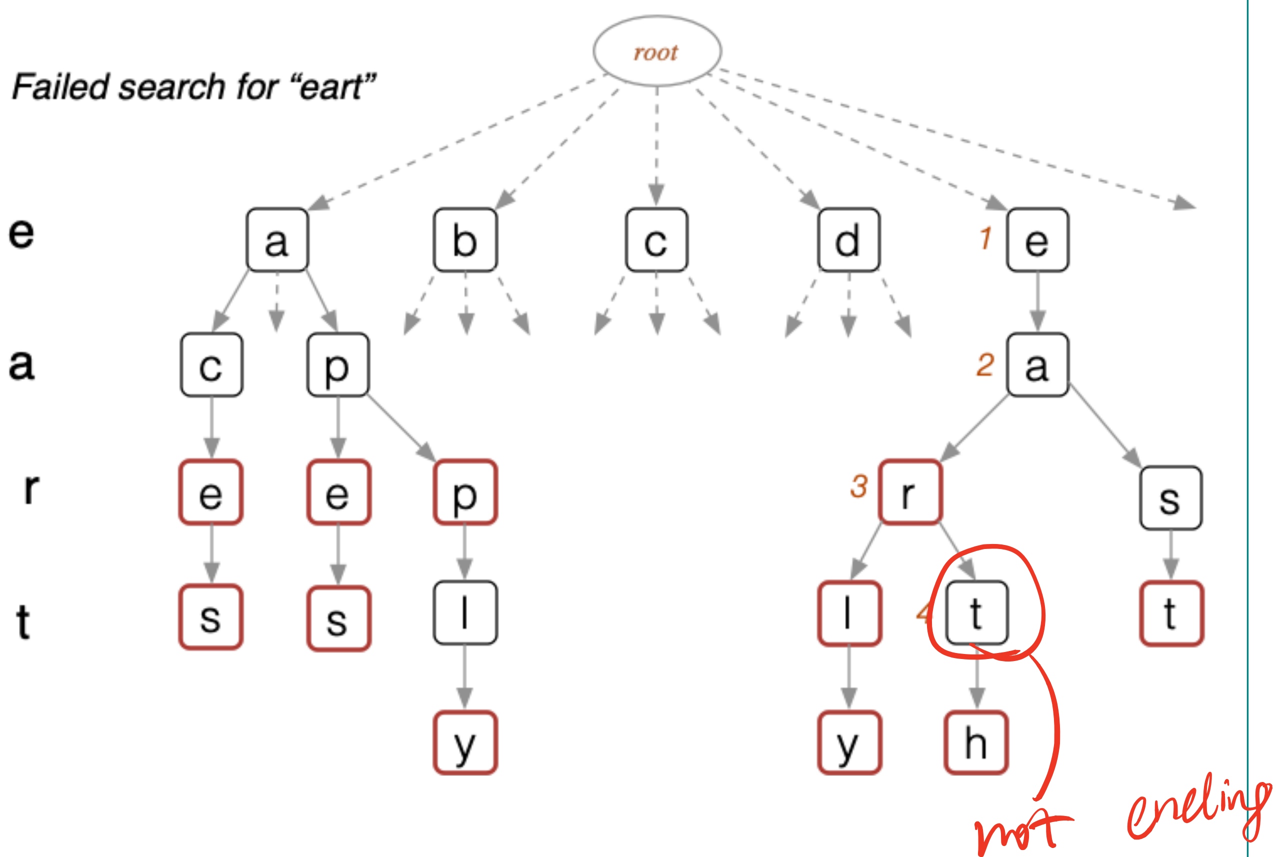
Search requires traversing a path, char-by-char from Key:

```
find(trie,key):
|  Input  trie, key
|  Output pointer to element in trie if key found
|         NULL otherwise
|
|  node=trie
|  for each char c in key do
|  |  if node.child[c] exists then
|  |     node=node.child[c]  // move down one level
|  |  else
|  |     return NULL
|  |  end if
|  end for
|  if node.finish then  // "finishing" node reached?
|     return node
|  else
|     return NULL
|  end if
```

❖ ... Searching in Tries



❖ ... Searching in Tries



❖ Insertion into Tries

Insertion into a Trie ...

```
Trie insert(trie,item,key):
|   Input  trie, item with key of length m
|   Output trie with item inserted
|
|   if trie is empty then
|   |   t=new trie node
|   end if
|   if m=0 then   // end of key
|   |   t.finish=true, t.data=item
|   else
|   |   first=key[0],  rest=key[1..m-1]
|   |   t.child[first]=insert(t.child[first],item,rest)
|   end if
|   return t
```

❖ Cost Analysis

Analysis of standard trie:

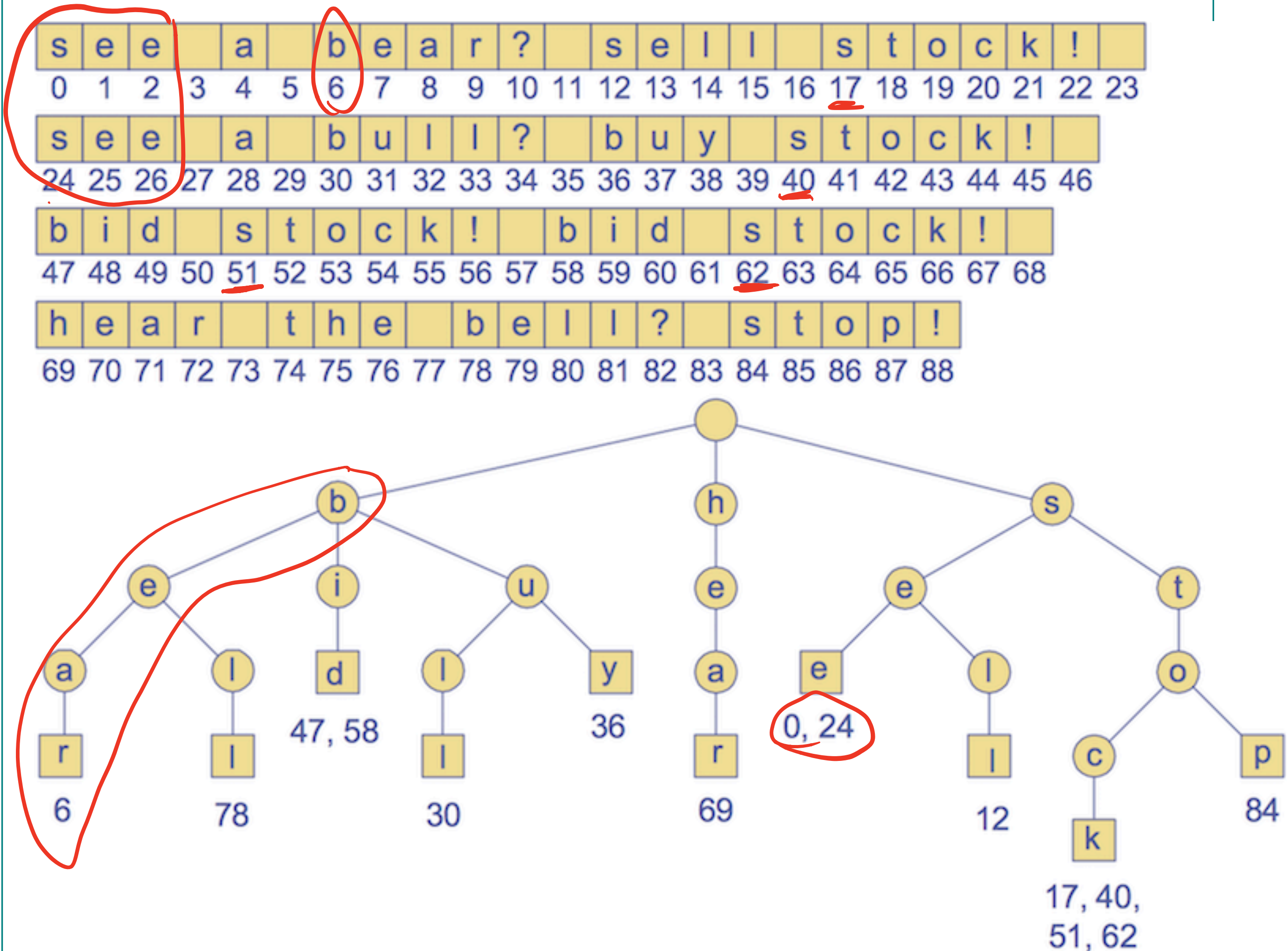
- *O(n)* space
- *O(m)* insertion and search

where

- *n* ... total size of text (e.g. sum of lengths of all strings)
- *m* ... length of the key string
- *d* ... size of the underlying alphabet (e.g. 26)

❖ Example Trie

Example text and corresponding trie of searchable words:



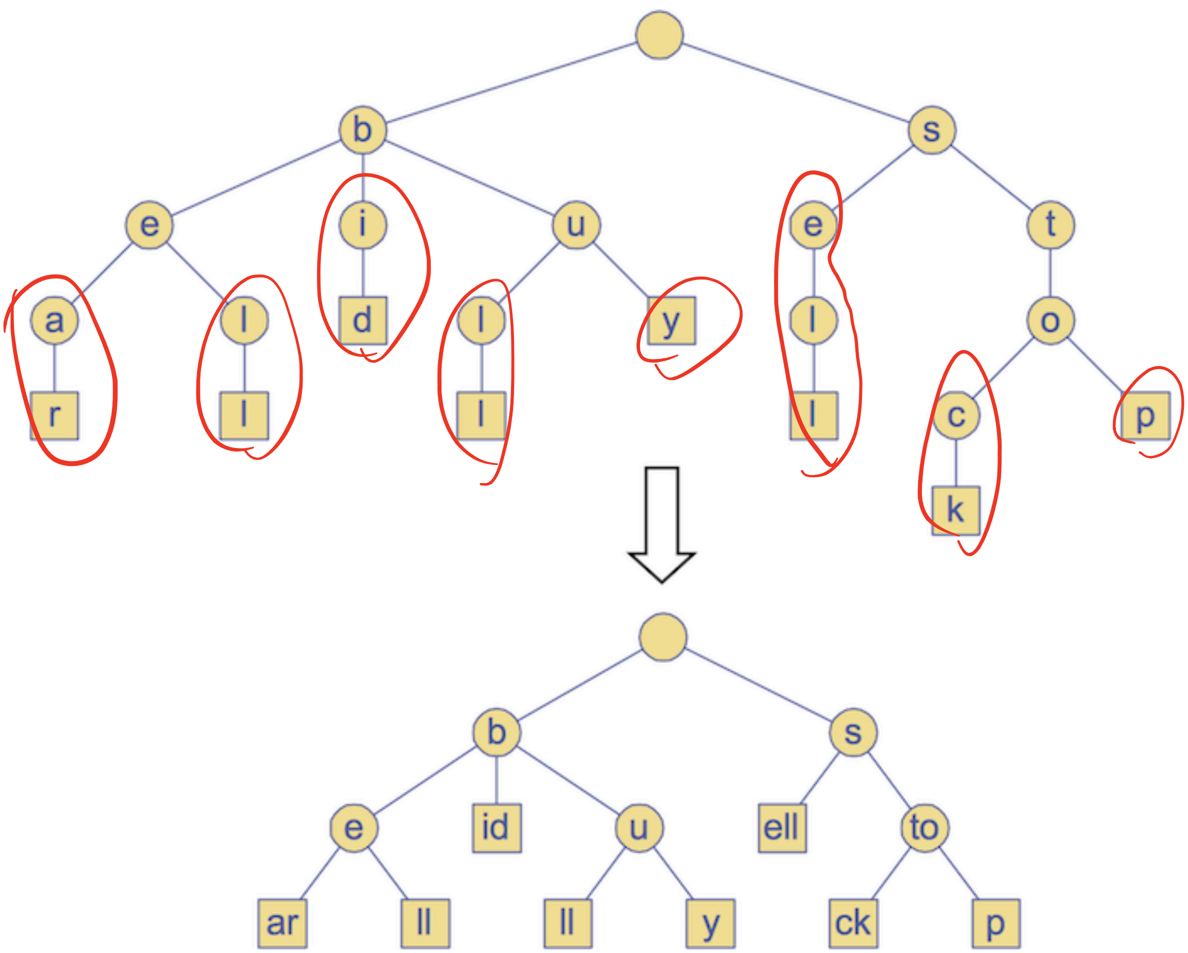
Note: trie has no prefixes ⇒ all finishing nodes are leaves

❖ Compressed Tries

Compressed tries ...

- have internal nodes of degree ≥ 2; each node contains ≥ 1 char
- obtained by compressing non-branching chains of nodes

Example:



❖ ... Compressed Tries

Compact representation of compressed trie to encode array *S* of strings:

- nodes store ranges of indices instead of substrings
    - use triple *(i,j,k)* to represent substring *S[i][j..k]*
- requires *O(s)* space (*s* = #strings in array *S*)

Example: