CHAPTER 8

HASHING

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed

"Fundamentals of Data Structures in C",

Symbol Table

- Definition

  A set of name-attribute pairs

- Operations
  - Determine if a particular name is in the table
  - Retrieve the attributes of the name
  - Modify the attributes of that name
  - Insert a new name and its attributes
  - Delete a name and its attributes

The ADT of Symbol Table

Structure SymbolTable(SymTab) is
objects: a set of name-attribute pairs, where the names are unique
functions:
for all *name* belongs to *Name*, *attr* belongs to *Attribute*, *symtab* belongs to *SymbolTable*, *max_size* belongs to integer

SymTab Create(max_size) ::= create the empty symbol table whose maximum capacity is *max_size*

Boolean IsIn(symtab, name) ::= if (name is in symtab) return TRUE
else return FALSE

Attribute Find(symtab, name) ::= if (name is in symtab) return the corresponding attribute else return null attribute

SymTab Insert(symtal, name, attr) ::= if (name is in symtab)
replace its existing attribute with attr
else insert the pair (name, attr) into
symtab

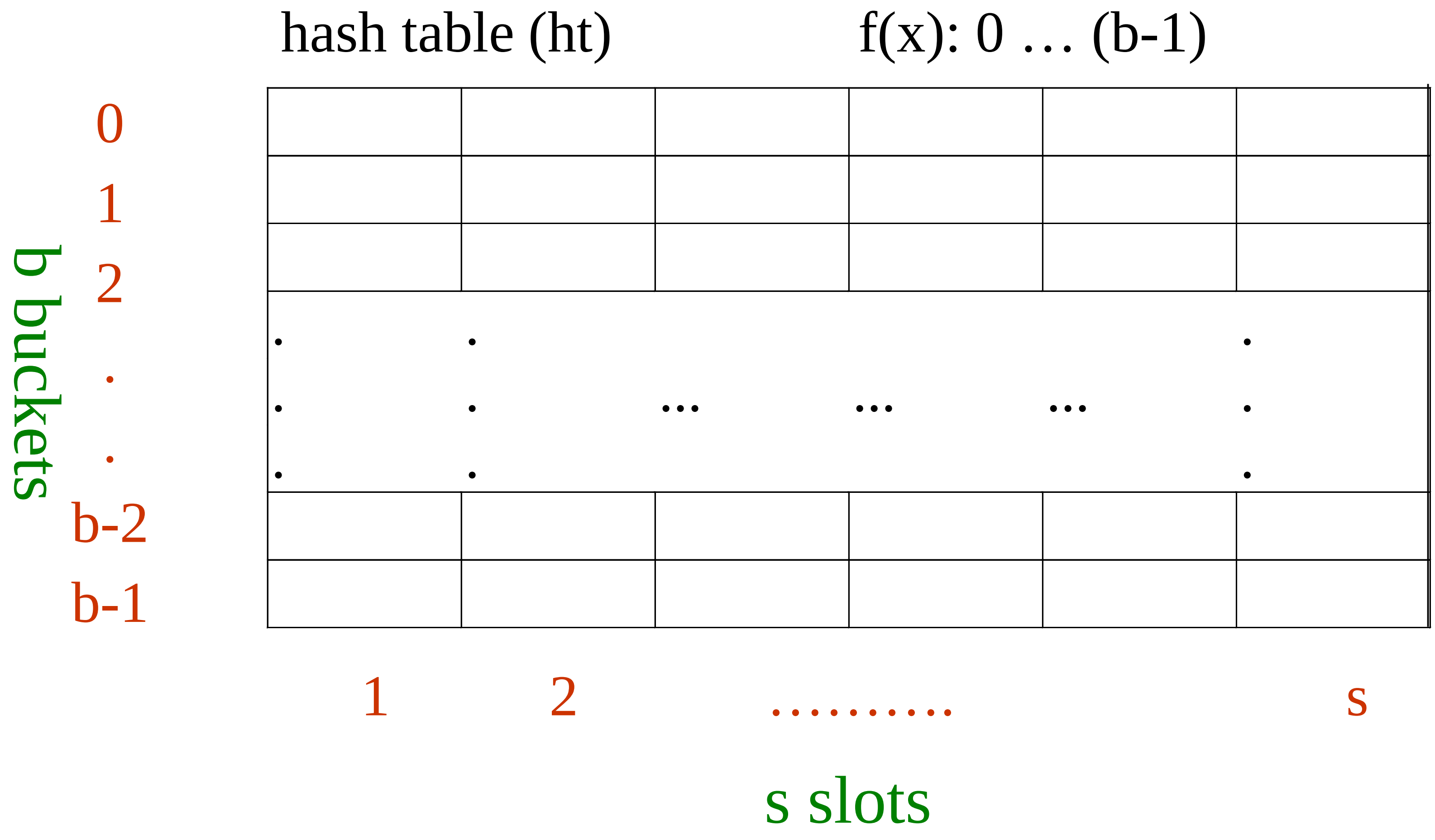
SymTab Delete(symtab, name) ::= if (name is not in symtab) return
else delete (name, attr) from symtab

search, insertion, deletion

Search vs. Hashing

- Search tree methods: key comparisons
- Hashing methods: hash functions
- types
  - statistic hashing
  - dynamic hashing

Static Hashing



Identifier Density and Loading Density

- The *identifier density* of a hash table is the ratio n/T
  - n is the number of identifiers in the table
  - T is possible identifiers
- The *loading density* or *loading factor* of a hash table is $\alpha = n/(sb)$
  - s is the number of slots
  - b is the number of buckets

Synonyms

- Since the number of buckets b is usually several orders of magnitude lower than T, the hash function f must map several different identifiers into the same bucket
- Two identifiers, i and j are synonyms with respect to f if $f(i) = f(j)$

Overflow and Collision

- An overflow occurs when we hash a new identifier into a full bucket

- A collision occurs when we hash two non-identical identifiers into the same bucket

Example

	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
…		
25		

synonyms (row 0: acos, atan)

synonyms (row 5: float, floor)

synonyms:
char, ceil,
clock, ctime

↑

overflow

b=26, s=2, n=10, α=10/52=0.19, f(x)=the first char of x

x: acos, define, float, exp, char, atan, ceil, floor, clock, ctime

f(x):0, 3, 5, 4, 2, 0, 2, 5, 2, 2

Hashing Functions

- Two requirements
  1. easy computation
  2. minimal number of collisions
- mid-square (middle of square)

$$f_m(x) = middle(x^2)$$

- division

$$f_D(x) = x\%M \qquad (0 \sim (M-1))$$

Avoid the choice of M that leads to many collisions

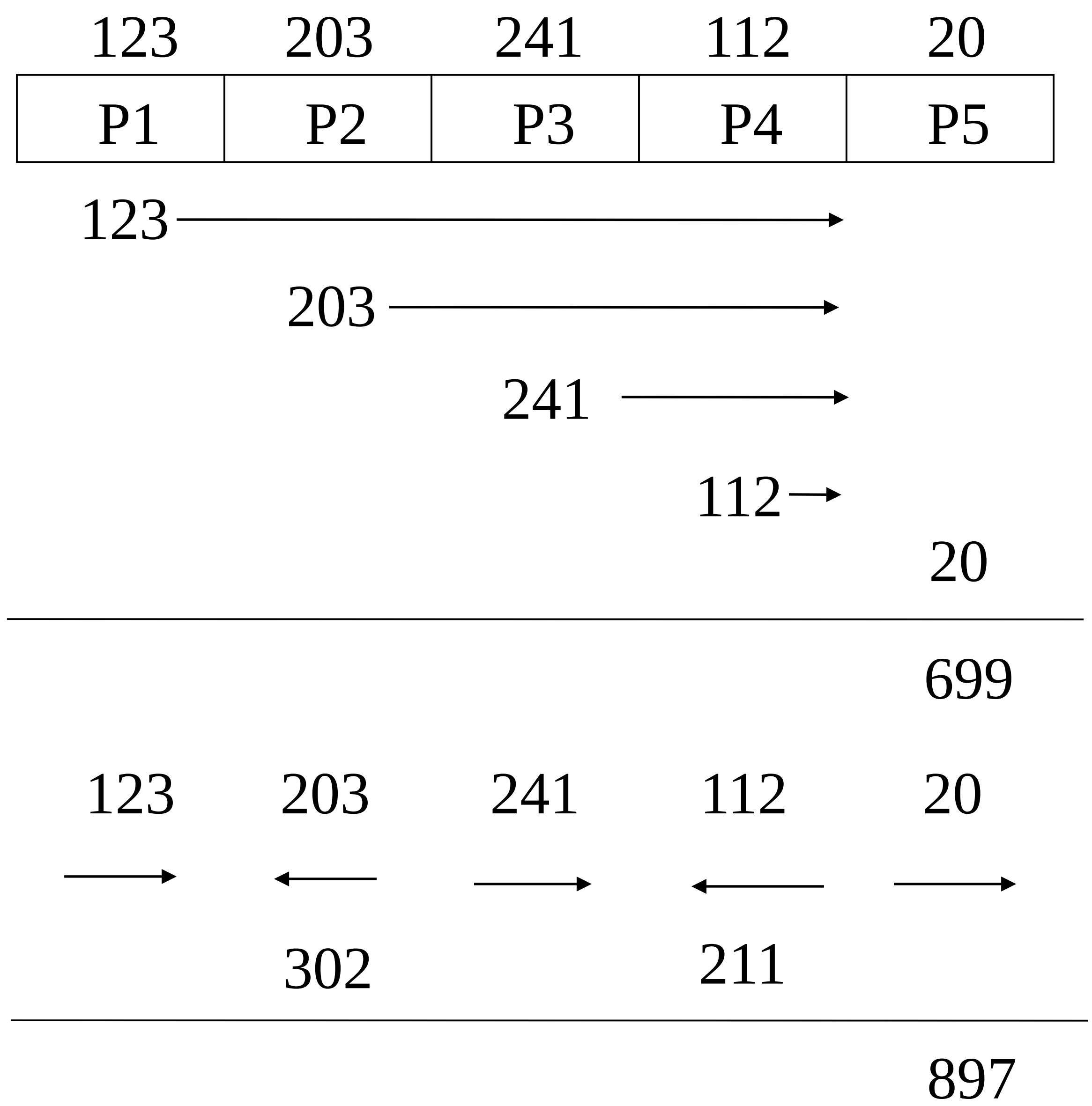
Hashing Functions

- Folding
  - Partition the identifier x into several parts
  - All parts except for the last one have the same length
  - Add the parts together to obtain the hash address
  - K=12320324111220
  - Two possibilities
    - Shift folding
      - x1=123, x2=203, x3=241, x4=112, x5=20, address=699
    - Folding at the boundaries
      - x1=123, x2=203, x3=241, x4=112, x5=20, address=897

shift folding

folding at
the boundaries

MSD ---> LSD
LSD <--- MSD



Digital Analysis

- All the identifiers are known in advance

  $M = 1 \sim 999$

X_1	d_{11}	d_{12}	…	d_{1n}
X_2	d_{21}	d_{22}	…	d_{2n}
…				
X_m	d_{m1}	d_{m2}	…	d_{mn}

  Select 3 digits from n

  Criterion:

  Delete the digits having the most skewed distributions

Overflow Handling

- Linear Open Addressing (linear probing)
- Quadratic probing
- Chaining

Data Structure for Hash Table

```
#define MAX_CHAR  10
#define TABLE_SIZE  13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

Example

Identifier	Additive Transform	x	Hash
for	102+111+114	327	2
do	100+111	211	3
while	119+104+105+108	537	4
if	105+102	207	12
else	101+108+115+101	425	9
function	102+117+110+99+	870	12

[0]	function
[1]	
[2]	for
[3]	do
[4]	while
[5]	
[6]	
[7]	
[8]	
[9]	else
[10]	
[11]	
[1 英	if

Linear Probing (linear open addressing)

- Compute f(x) for identifier x
- Examine the buckets

  ht[(f(x)+j)%TABLE_SIZE]

  $0 \le j \le$ TABLE_SIZE

  - The bucket contains x.
  - The bucket contains the empty string
  - Return to ht[f(x)]

桶	x	桶的搜尋次數
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
…		
25		

Linear Probing

```
void linear_insert(element item, element ht[])
{
  int i, hash_value;
  i = hash_value = hash(item.key);
  while(strlen(ht[i].key)) {
  if (!strcmp(ht[i].key, item.key))
      fprintf(stderr, “Duplicate entry\n”);
      exit(1);
    }
    i = (i+1)%TABLE_SIZE;
    if (i == hash_value) {
      fprintf(stderr, “The table is full\n”);
      exit(1);
    }
  }
  ht[i] = item;
}
```

Problem of Linear Probing

- Identifiers tend to cluster together
- Adjacent cluster tend to coalesce
- Increase the search time

Quadratic Probing

- Linear probing searches buckets $(f(x)+i)\%b$
- Quadratic probing uses a quadratic function of i as the increment
- Examine buckets $f(x)$, $(f(x)+i^2)\%b$, $(f(x)-i^2)\%b$, for $1 \le i \le (b-1)/2$
- b is a prime number of the form $4j+3$, j is an integer

Rehashing

- Try f_1, f_2, …, f_m in sequence if collision occurs
- disadvantage
  - comparison of identifiers with different hash values
  - use chain to resolve collisions

Data Structure for Chaining

```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
typedef struct list *list_pointer;
typedef struct list {
    element item;
    list_pointer link;
};
list_pointer hash_table[TABLE_SIZE];
```

Chain Insert

```
void chain_insert(element item, list_pointer ht[])
{
  int hash_value = hash(item.key);
  list_pointer ptr, trail=NULL, lead=ht[hash_value];
  for (; lead; trail=lead, lead=lead->link)
    if (!strcmp(lead->item.key, item.key)) {
      fprintf(stderr, “The key is in the table\n”);
      exit(1);
  }
 ptr = (list_pointer) malloc(sizeof(list));
   if (IS_FULL(ptr)) {
     fprintf(stderr, “The memory is full\n”);
     exit(1);
   }
   ptr->item = item;
   ptr->link = NULL;
   if (trail) trail->link = ptr;
   else ht[hash_value] = ptr;
 }
```

Results of Hash Chaining

acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime
f(x)=first character of x

[0]	-> acos -> atoi -> atol
[1]	-> NULL
[2]	-> char -> ceil -> cos -> ctime
[3]	-> define
[4]	-> exp
[5]	-> float -> floor
[6]	-> NULL
…	
[25]	-> NULL

# of key comparisons=21/11=1.91

The *loading density* or *loading factor* of a hash table is $\alpha = n/(sb)$

α=n/b	.50	.75	.90	.95
hashing function	chain/open	chain/open	chain/open	chain/open
mid square	1.26/1.73	1.40/9.75	1.45/37.14	1.47/37.53
division	1.19/4.52	1.31/7.20	1.38/22.42	1.41/25.79
shift fold	1.33/21.75	1.48/65.10	1.40/77.01	1.51/118.57
Bound fold	1.39/22.97	1.57/48.70	1.55/69.63	1.51/97.56
digit analysis	1.35/4.55	1.49/30.62	1.52/89.20	1.52/125.59
theoretical	1.25/1.50	1.37/2.50	1.45/5.50	1.48

n is the number of identifiers in the table

b is the number of buckets

Dynamic Hashing (extensible hashing)

- Dynamically increasing and decreasing file size
- concepts
  - file: a collection of records
  - record: a key + data, stored in pages (buckets)
  - space utilization

$$\frac{Numberof\ \mathrm{Re}cord}{NumberofPages * PageCapacity}$$

Dynamic Hashing Using Directories

Example. m(# of pages)=4, P(page capacity)=2

00, 01, 10, 11

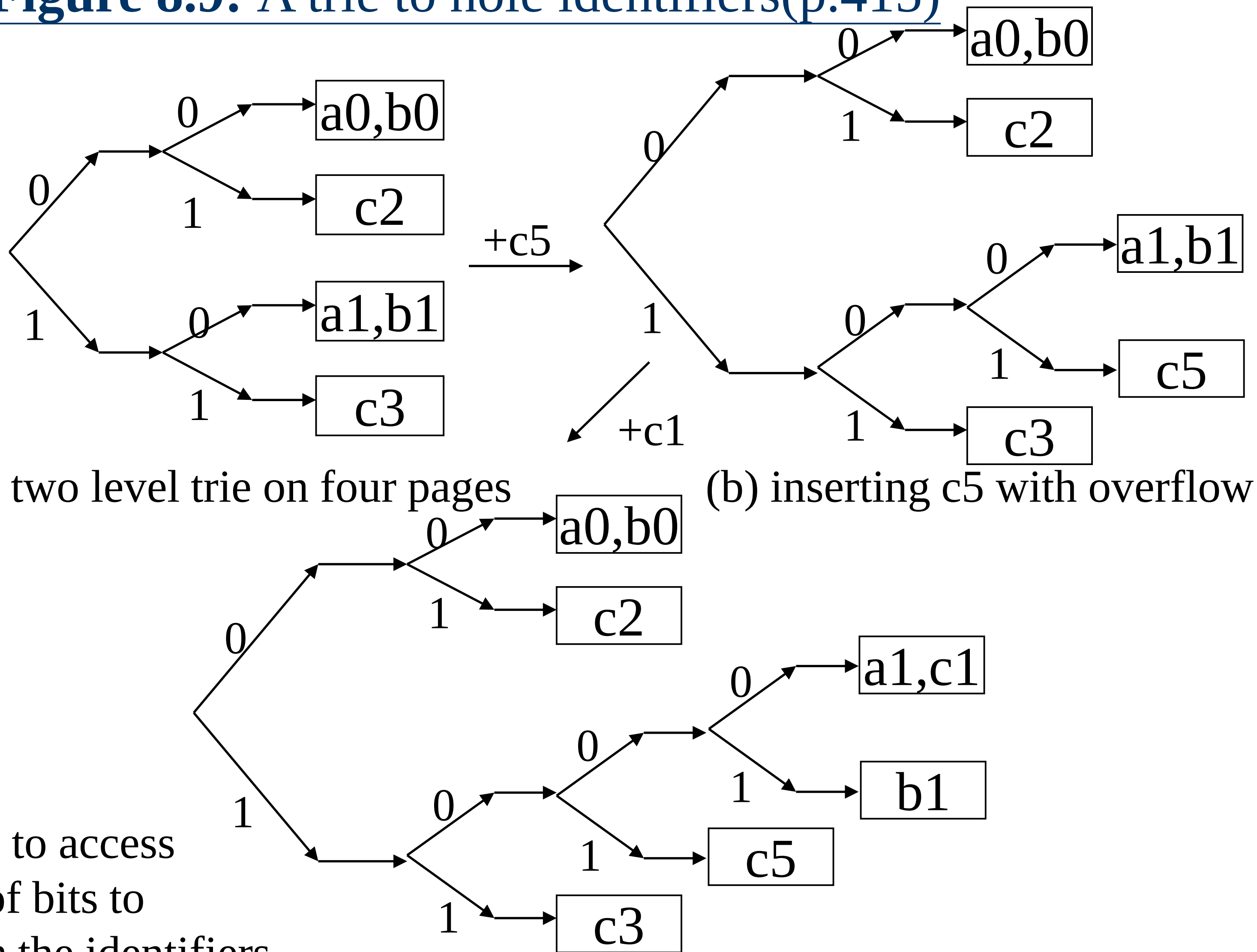
Identifiers	Binary representaiton
a0	100 0<u>00</u>
a1	100 0<u>01</u>
b0	101 0<u>00</u>
b1	101 0<u>01</u>
c0	110 0<u>00</u>
c1	110 0<u>01</u>
c2	110 0<u>10</u>
c3	110 0<u>11</u>

from LSB to MSB

allocation: lower order two bits

***Figure 8.8:** Some identifiers requiring 3 bits per character(p.414)

***Figure 8.9:** A trie to hole identifiers(p.415)*



(a) two level trie on four pages

(b) inserting c5 with overflow

(c) inserting c1 with overflow

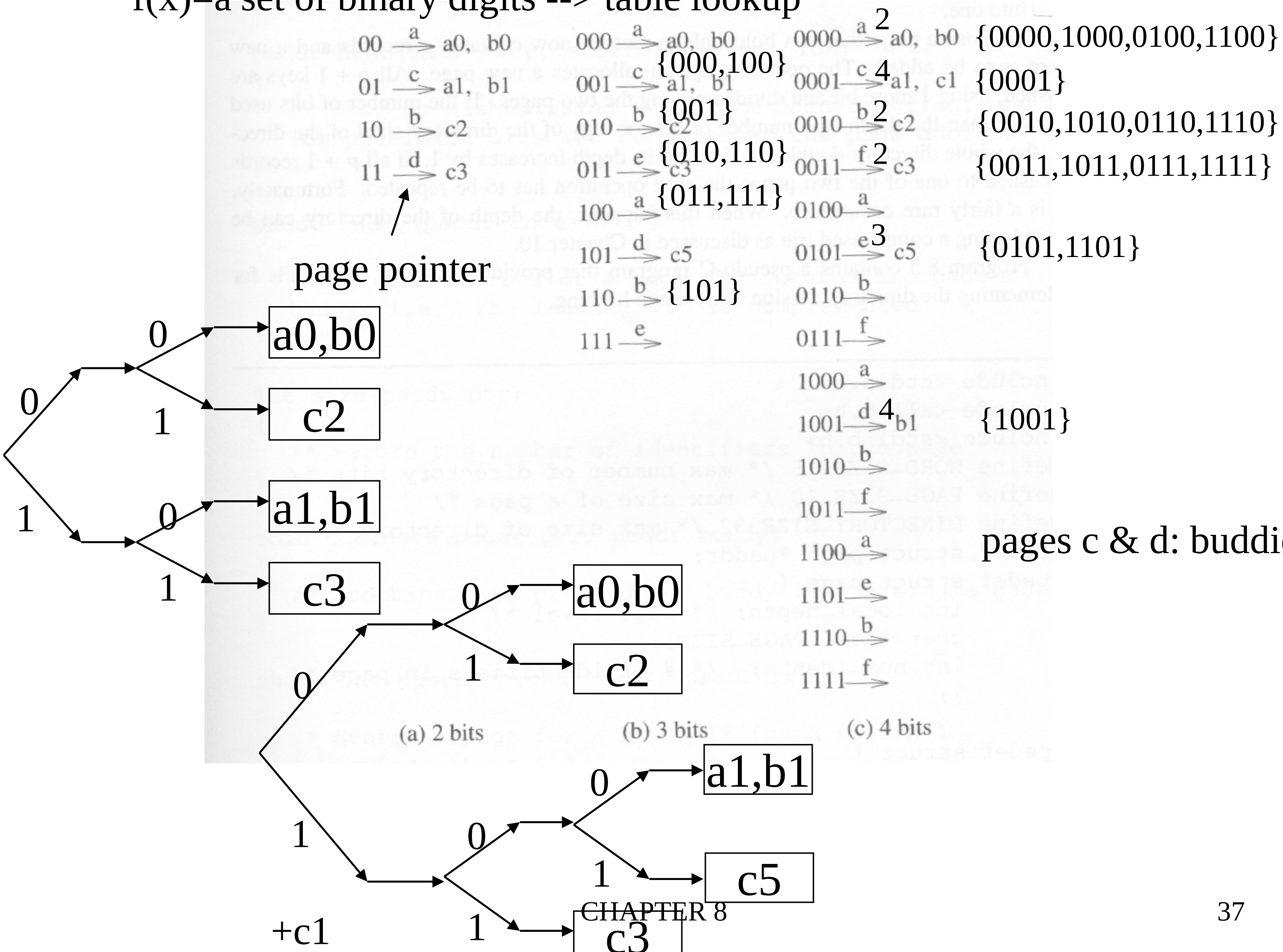
Note: time to access a page: # of bits to distinguish the identifiers

Note: identifiers skewed: depth of tree skewed

Extendiable Hashing

f(x)=a set of binary digits --> table lookup

local depth
global depth: 4



pages c & d: buddies

If keys do not uniformly divide up among pages, then the directory can glow quite large, but most of entries will point to the same page

f: a family of hashing functions

$hash_i$: key --> {0 .. 2^{i-1}} $1 \le i \le d$

hash(key, i): produce random number of i bits from identifier key

$hash_i$ is $hash_{i-1}$ with either a zero or one appeared as the new leading bit of result

100 0$\underline{00}$	10$\underline{0\ 001}$	101 0$\underline{00}$
hash(a0,2)=00	hash(a1,4)=0001	hash(b0,2)=00
10$\underline{1\ 001}$	11$\underline{0\ 001}$	110 0$\underline{10}$
hash(b1,4)=1001	hash(c1, 4)=0001	hash(c2, 2)=10
110 0$\underline{11}$	110 $\underline{101}$	
hash(c3,2)=11	hash(c5,3)=101	

*Program 8.5: Dynamic hashing (p.421)

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>      2^5=32
#define WORD_SIZE 5        /* max number of directory bits */
#define PAGE_SIZE 10       /* max size of a page */
#define DIRECTORY_SIZE 32   /* max size of directory */
typedef struct page *paddr;
typedef struct page {                    See Figure 8.10(c) global depth=4
                                                           local depth of a =2
        int local_depth; /* page level */
        char  *name[PAGE_SIZE];     the actual identifiers
        int num_idents;   /* #of identifiers in page */
        };
typedef struct {
        char  *key;        /* pointer to string */
        /*other fields */
        } brecord;
int global_depth; /* trie height */
paddr directory[DIRECTORY_SIZE]; /* pointers to pages  */
```

```
paddr hash(char *, short int);
paddr buddy(paddr);
short int pgsearch(char *, paddr );
int convert(paddr);
void enter(brecord, paddr);
void pgdelete(char *, paddr);
paddr find(brecord, char *);
void insert (brecord, char *);
int size(paddr);
void coalesce (paddr, paddr);
void delete(brecord, char *);

paddr hash(char *key, short int precision)
{
  /*    *key is hashed using a uniform hash function, and the
        low precision bits are returned as the page address */
 }
```

directory subscript for directory lookup

```
paddr buddy(paddr index)
{
 /*Take an address of a page and returns the page's
    buddy, i. e., the leading bit is complemented */
 }
```

buddy $b_{n-1}b_{n-2} \ldots b_0$ $\qquad$ $\overline{b_{n-1}}b_{n-2} \ldots b_0$

```
 int size(paddr ptr)
 {
   /* return the number of identifiers in the page */
 }
 void coalesce(paddr ptr, paddr, buddy)
{
  /*combine page ptr and its buddy into a single page */
 }
short int pgsearch{char *key, paddr index)
 {
    /*Search a page for a key. If found return 1
    otherwise return 0 */
}
```

```
void convert (paddr ptr)
{
 /* Convert a pointer to a pointer to a page to an equivalent integer */
}

void enter(brecord r, paddr ptr)
{
   /* Insert a new record into the page pointed at by ptr */
}

void pgdelete(char *key, paddr ptr)
{
  /* remove the record with key, hey, from the page pointed to by ptr */
}

short int find (char *key, paddr *ptr)
{
   /* return 0 if key is not found and 1 if it is. Also,
     return a pointer (in ptr) to the page that was searched.
     Assume that an empty directory has one page. */
```

```
paddr   index;
int intindex;
index = hash(key, global_depth);
intindex = convert(index);
*ptr = directory[intindex];
return pgsearch(key, ptr);
}

 void insert(brecord r, char *key)
 {
  paddr ptr;
  if find(key, &ptr) {
        fprintr(stderr, " The key is already in the table.\n");
        exit(1);
  }
  if (ptr-> num_idents != PAGE_SIZE) {
     enter(r, ptr);
     ptr->num_idents++;
 }
 else{ /*Split the page into two, insert the new key, and update global_depth
             if necessary.
```

```
  If this causes global_depth to exceed WORD_SIZE then print an error
  and terminate. */
 };
}

void delete(brecord r, char *key)
{
/* find and delete the record r from the file */
   paddr  ptr;
   if (!find (key, &ptr )) {
       fprintf(stderr, “Key is not in the table.\n”);
       return; /* non-fatal error */
   }
   pgdelete(key, ptr);
   if (size(ptr) + size(buddy(ptr)) <= PAGE_SIZE)
        coalesce(ptr, buddy(ptr));
}

void main(void)
{
}
```

Directoryless Dynamic Hashing (Linear Hashing)

continuous address space

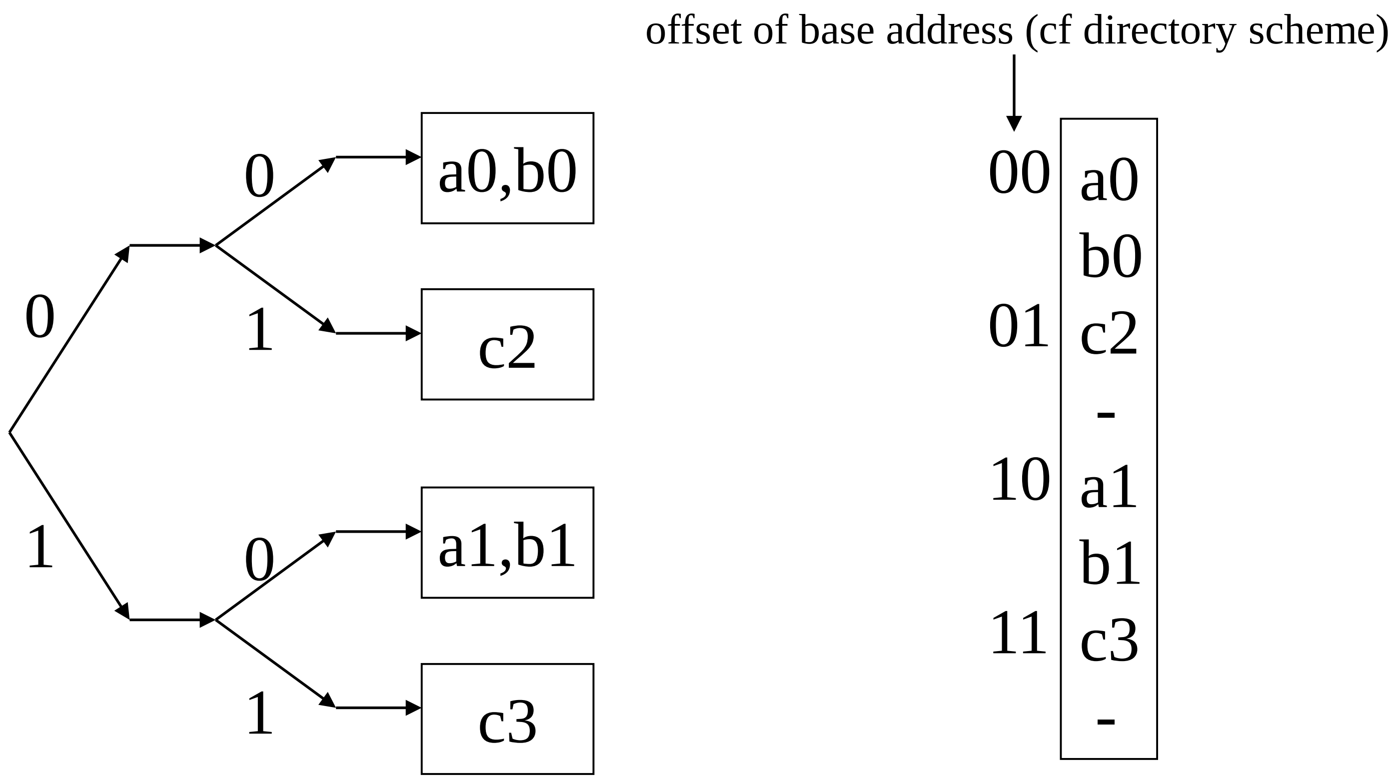
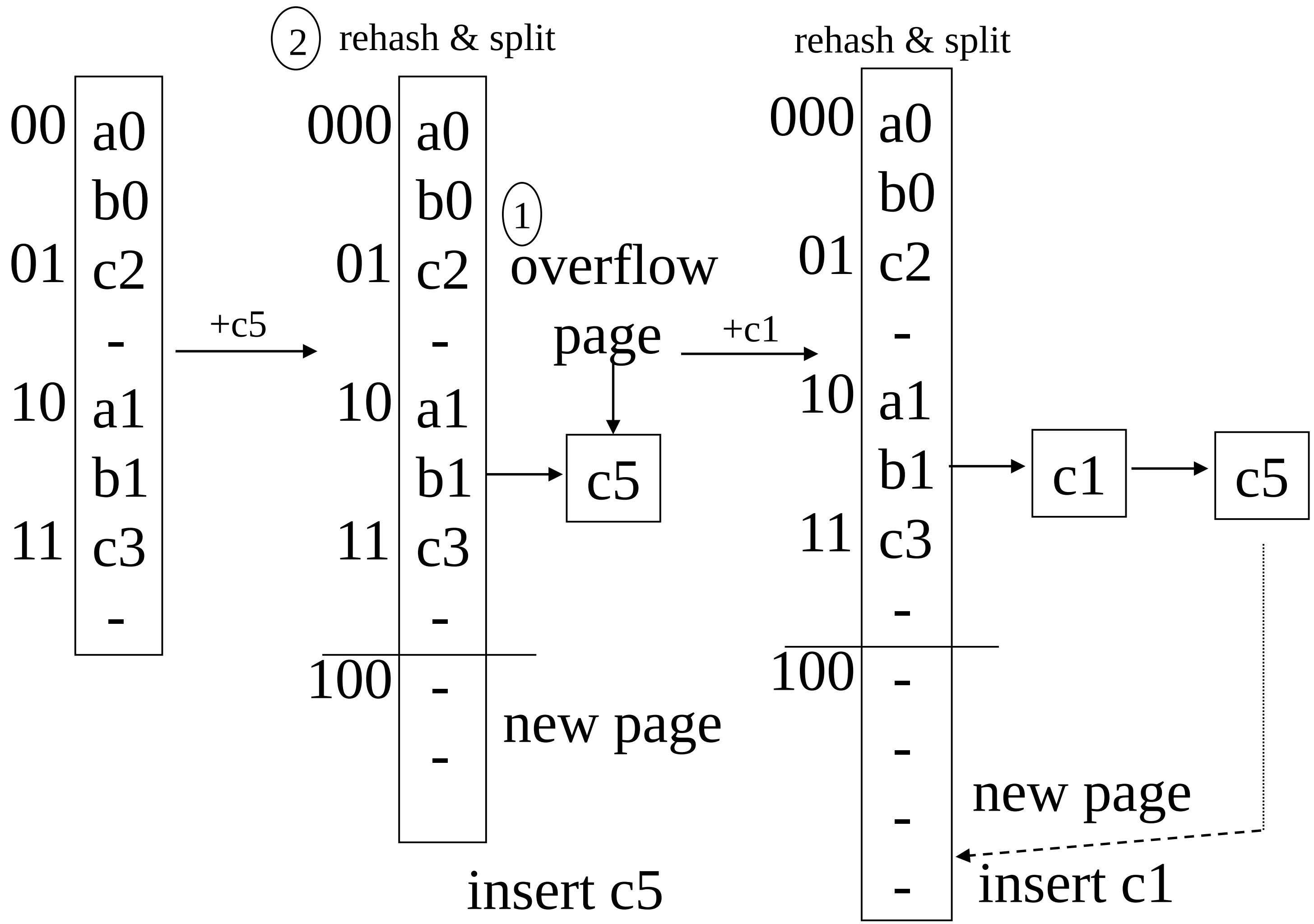


***Figure 8.12:** A trie mapped to a directoryless, contiguous storage (p.424)

*Figure 8.13: An example with two insertions (p.425)

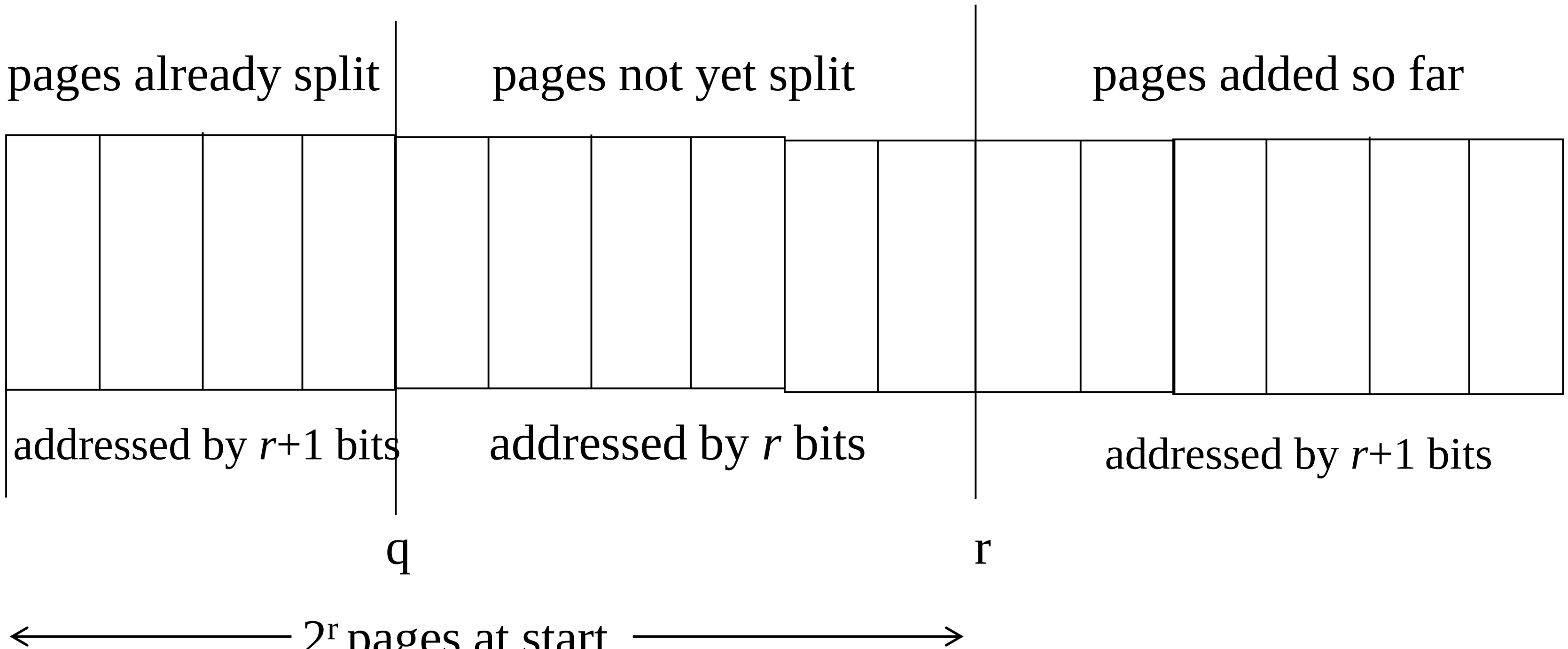


start of expansion 2
there are 4 pages
(a)

insert c5
page 10 overflows
page 00 splits
(b)

page 10 overflows
page 01 splits
(c)

***Figure 8.14:** During the *r*th phase of expansion of directoryless method (p.426)



suppose we are at phase r; there are 2^r pages indexed by *r* bits

***Program 8.6:**Modified hash function (p.427)

```
if ( hash(key,r) < q)
      page = hash(key, r+1);
  else
      page = hash(key, r);
```
if needed, then follow overflow pointers;