# Hashing

- 8.1 Introduction
- 8.2 Static Hashing
  - Hash Tables
  - Hashing Functions
  - Overflow Handling
    - Open addressing
    - Chaining
- 8.3 Dynamic Hashing

# Chaining

| bucket | $x$ | buckets searched |
|---|---|---|
| 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
  - Perform poorly;
  - The search for a key involves comparison of identifiers with different hash values : ex) atol → …
- Chained hash table
  - One list per bucket
  - Each list containing all the synonyms for that bucket
  - Searching a key *k*
    - Compute *h*(k)
    - Examine only those keys in the list for *h*(k)
  - Use array & chains

[0] → **acos atoi atol**
[1] → *NULL*
[2] → **char ceil cos ctime**
[3] → **define**
[4] → **exp**
[5] → **float floor**
[6] → *NULL*
...
[25] → *NULL*

**Figure 8.6:** Hash chains corresponding to Figure 8.4

ht[0:b-1]

[0] → **acos atoi atol**
[1] → *NULL*
[2] → **char ceil cos ctime**
[3] → **define**
[4] → **exp**
[5] → **float floor**
[6] → *NULL*
. . .
[25] → *NULL*

**Figure 8.6:** Hash chains corresponding to Figure 8.4

Average # of key comparisons for searching *k*
= 21/11 = **1.91**

```
element* search(int k)
{/* search the chained hash table ht for k,
if a pair with this key is found, return a pointer to this pair;
otherwise, return NULL. */
	nodePointer current;
	int homeBucket = h(k);

	/* search the chain ht[homeBucket] */
	for(current = ht[homeBucket]; current; current = current->link)
		if(current->data.key == k) return &current->data;
	return NULL;
}
```

Program 8.4: Chain search

[0] → **acos atoi atol**
[1] → *NULL*
[2] → **char ceil cos ctime**
[3] → **define**
[4] → **exp**
[5] → **float floor**
[6] → *NULL*
...
[25] → *NULL*

**Figure 8.6:** Hash chains corresponding to Figure 8.4

- chaining + uniform hash function
  - The expected average number of key comparisons for a successful search:
    $\approx 1+\alpha/2$, where $\alpha = n/b$
  - Ex) $\alpha = 0.5 \rightarrow 1.25$, $\alpha = 1 \rightarrow 1.5$
  - Performance
    - Depends only on the method used to handle **overflows** (when the keys are selected at random from the key space)
    - In practice, a tendency to make a biased use of **keys**
    - Different **hash functions** result in different performance
    - Generally, the division hash function coupled with chaining yields best performance
  - The worst-case for a successful search
    - $O(n)$
    - Reduced to $O(\log n)$ by sorting synonyms in a balanced search tree

# 8.3 DYNAMIC HASHING

# Motivation for Dynamic Hashing

- Disadvantages of static hashing
  - … 데이터의 동적 삽입 삭제가 어렵다
    충돌이 발생하면 처리방법이 제한적이고, 성능 저하되기가 쉽다
- Dynamic (Extendible) hashing
  - Reduce the rebuild time Rebuild 할때 시간을 줄인다
  - Each rebuild changes the home bucket for the entries in only 1 bucket
  - Two forms of dynamic hashing:
    Using **directory**, **directoryless**

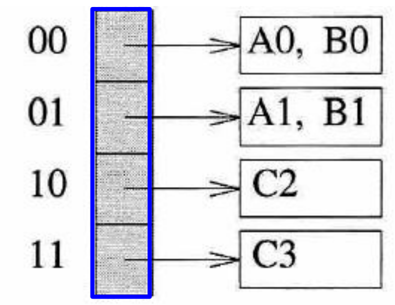
| $k$ | $h(k)$ |
|---|---|
| A0 | 100 000 |
| A1 | 100 001 |
| B0 | 101 000 |
| B1 | 101 001 |
| C1 | 110 001 |
| C2 | 110 010 |
| C3 | 110 011 |
| C5 | 110 101 |

**Figure 8.7:** An example hash function

***h*(*k*, *p*)** denotes the integer formed by the p LSBs of $h(k)$

$h(\text{A0},1) = 0$
$h(\text{A1},3) = 1$
$h(\text{B1},4) = 1001 = 9$

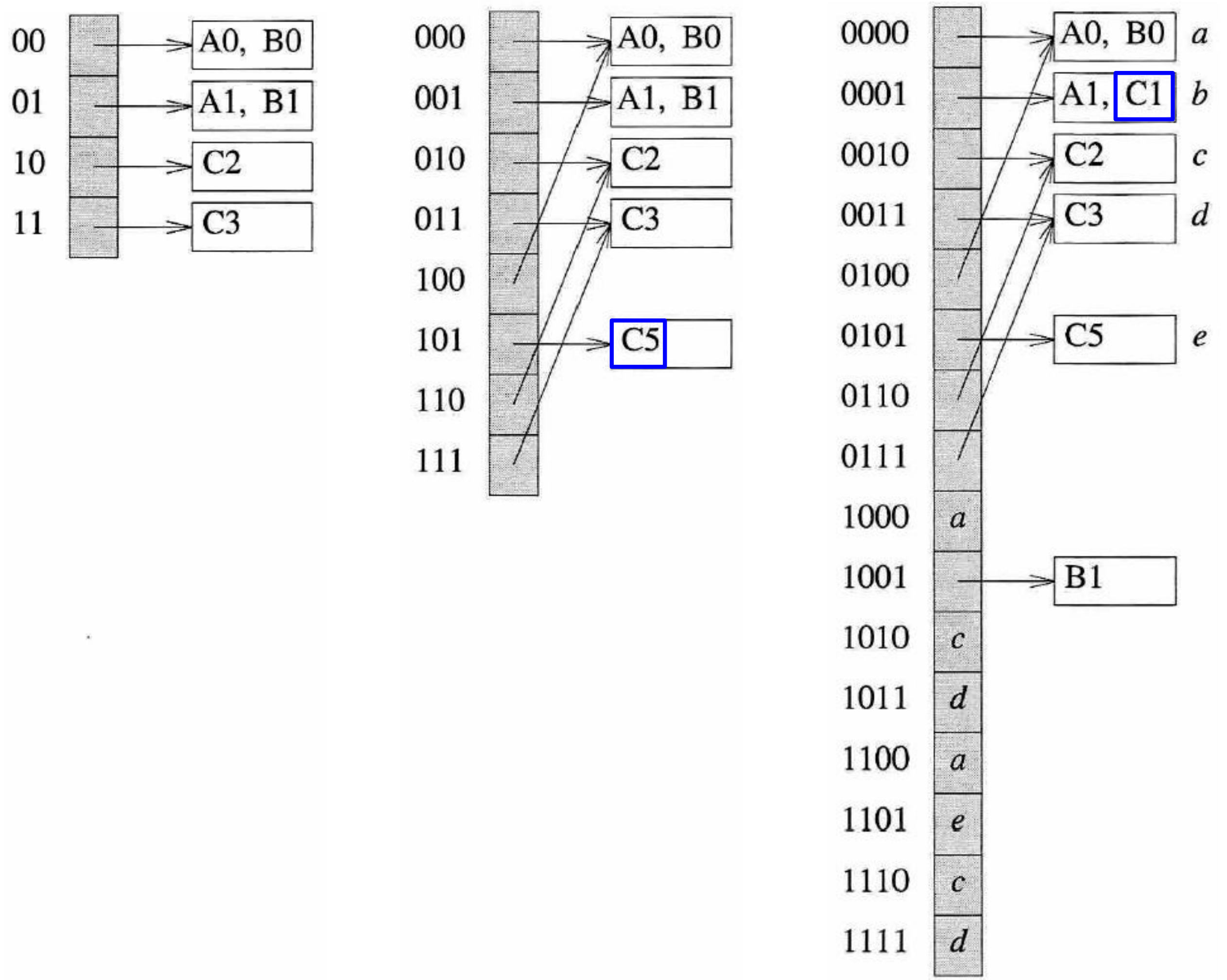
# Dynamic Hashing using Directories




directory

- Use a directory *d*: pointers to buckets
- The size of directory
  - Depend on the # of bits of $h(k)$ 비트수에 따라 size가 달라짐
  - $h(k,2)$: directory size = $2^2 = 4$
  - $h(k,5)$: $2^5 = 32$
- Directory depth
  - The # of bits of $h(k)$ used to index the directory 비트 수
- To search for a key k
  - Examine the bucket pointed to by d[$h(k$,t)]

| $k$ | $h(k)$ |
|---|---|
| A0 | 100 000 |
| A1 | 100 001 |
| B0 | 101 000 |
| B1 | 101 001 |
| C1 | 110 001 |
| C2 | 110 010 |
| C3 | 110 011 |
| C5 | 110 101 |




(a) depth = 2 (b) depth = 3 (c) depth = 4

**Figure 8.8:** Dynamic hash tables with directories

- Insert C5

  → $h(C5, 2) = 01$

  다찼어

  - Bucket overflow
  - Determine the least $u$ such that $h(k, u)$ is not the same for all keys in the overflowed bucket

    → $u = 3$ → $h(C5, 3) = 101$

  - In case $u > t$:

    1) increase directory depth to $u = 3$;

    → double the directory size

    → pointers in the original directory are duplicated

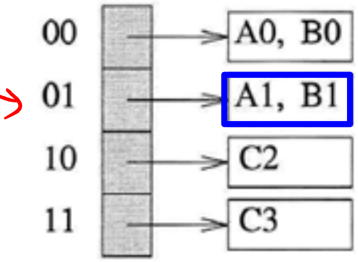
    원래있던 Point하는 유지함

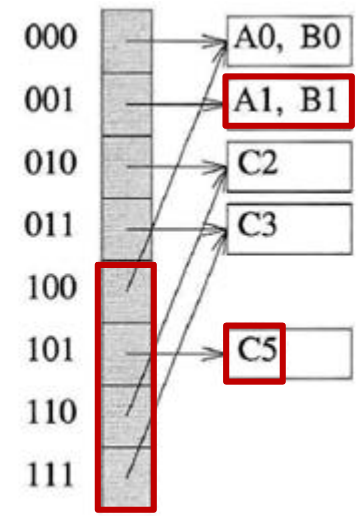
    2) split the overflowed bucket using $h(k, 3)$

    → 001: A1, B1, 101:C5

    → update d[101] to point to the new bucket

| $k$ | $h(k)$ |
|---|---|
| A0 | 100 000 |
| A1 | 100 001 |
| B0 | 101 000 |
| B1 | 101 001 |
| C1 | 110 001 |
| C2 | 110 010 |
| C3 | 110 011 |
| C5 | 110 101 |



- Insert C1

→ $h(\text{C1}, 2) = 01$

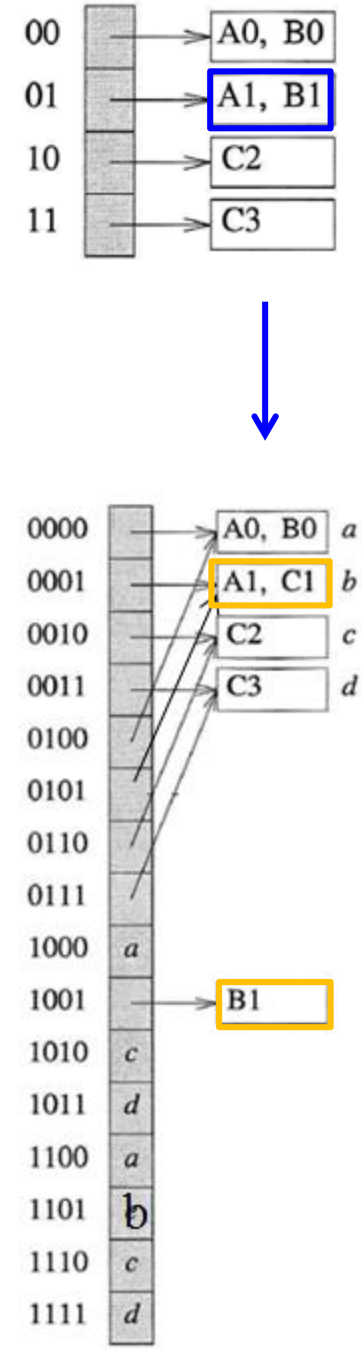
- Bucket overflow;
- Determine $u$

→ $u = 4$ (u=3은 여전히 overflow)

In case $u > t$:

1) increase directory depth to $u = 4$;
  → quadruple the directory size
  → pointers in the directory are replicated 3 times
2) split the overflowed bucket using $h(k, 4)$
  → 0001: A1, C1, 1001: B1
  → update d[1001] to point to the new bucket

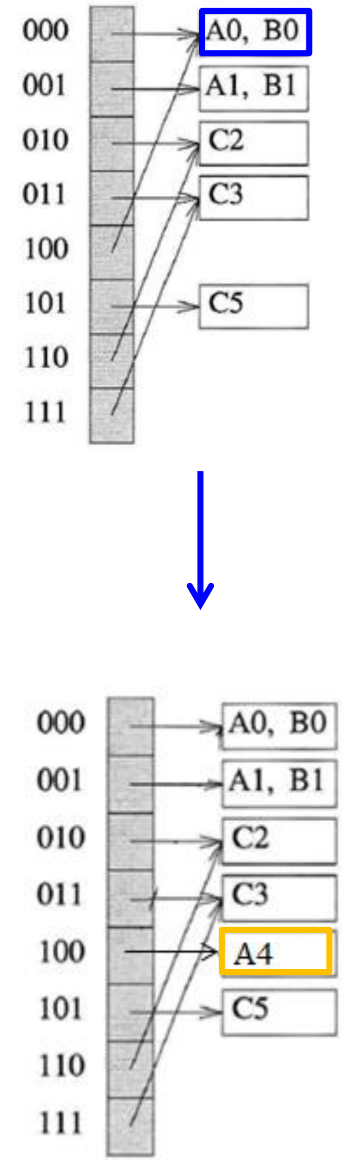
| $k$ | $h(k)$ |
|---|---|
| A0 | 100 000 |
| A1 | 100 001 |
| B0 | 101 000 |
| B1 | 101 001 |
| C1 | 110 001 |
| C2 | 110 010 |
| C3 | 110 011 |
| C5 | 110 101 |



- Insert A4

→ $h(\text{A4}, 3) = 100$
  - Bucket overflow;
  - Determine $u$
  - $u = 3$

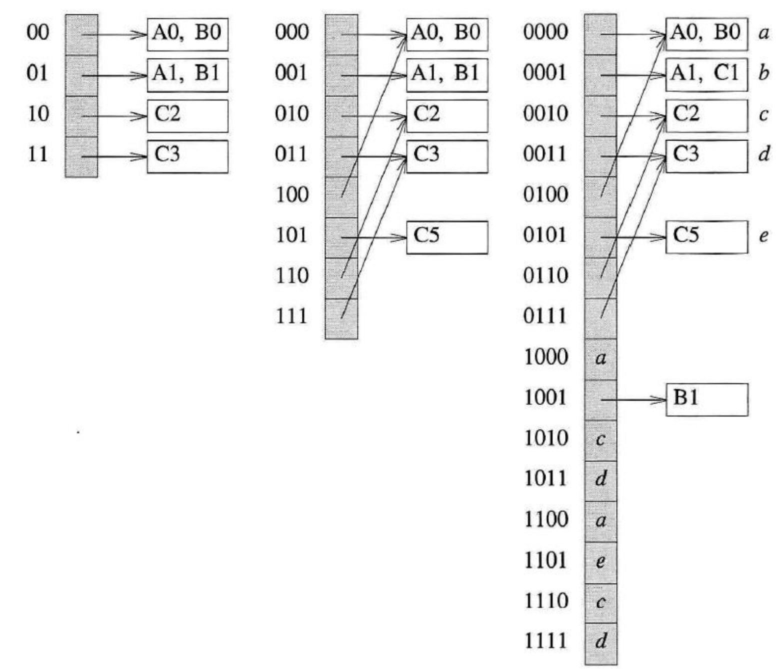
| $k$ | $h(k)$ |
|---|---|
| A0 | 100 000 |
| A1 | 100 001 |
| B0 | 101 000 |
| B1 | 101 001 |
| C1 | 110 001 |
| C2 | 110 010 |
| C3 | 110 011 |
| C5 | 110 101 |




In Case: $u <= t$
- The size of directory is not changed
- split the overflowed bucket using $h(k, u)$;
  - 000: A0, B0, 100:A4
- update d[100] to point to the new bucket

- Advantages
  - The time for array doubling is considerably less than that for the array doubling used in static hashing
  - Rehash only the entries in the bucket that overflows rather than all entries in the table

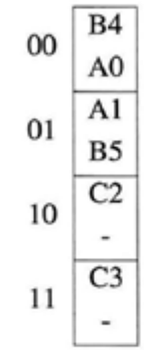
overflow 나는 애들만 rehash 하면된다




(a) depth = 2 (b) depth = 3 (c) depth = 4

**Figure 8.8:** Dynamic hash tables with directories

# Directoryless Dynamic Hashing




(a) $r = 2, q = 0$

- An array *ht* of buckets is used
- Assumption
  - Array is as large as possible; 충분한 크기의 배열을 이미 할당해놓음
  - There is no possibility of increasing its size dynamically 크기를 동적으로 늘릴 일이 없음
- Use two variables $q$ and $r$, $0 \leq q < 2^r$
  - Keep track of the active buckets 사용하고 있는 bucket
  - $r$: The # of bits of $h(k)$ used to index into the hash table
  - $q$: The bucket that will split next
- Only buckets $0 \sim 2^r+q-1$ are active
  - Each active bucket is the start of a chain of buckets

<Insert C5>

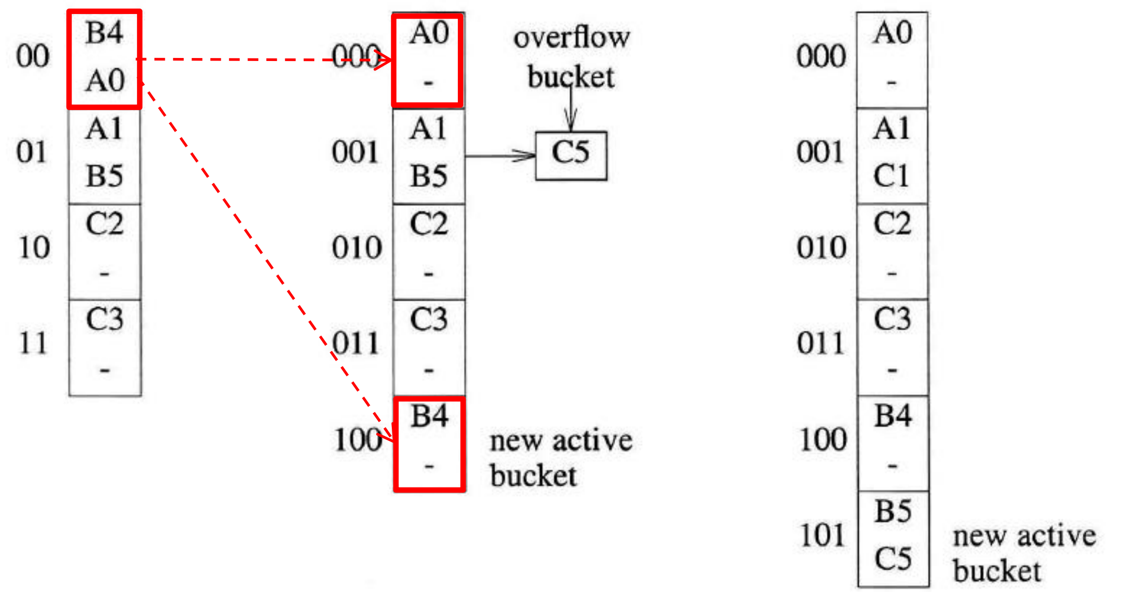
→ h(C5,2)=01

→ b[01]: overflow

→ activating bucket $2^r$+q;
reallocating entries in *q* by *h*(*k*, *r*+1)

→ *q*++

bucket 하나 더 쓰고
h(k,r+1)로 rehash




C5

$2^2$의 bucket

(a) $r = 2$, $q = 0$ (b) Insert C5, $r = 2$, $q = 1$ (c) Insert C1, $r = 2$, $q = 2$

**Figure 8.9:** Inserting into a directoryless dynamic hash table

<Insert C1>
→ h(C1,2)=01
→ d[01]: overflow
→ Activating $2^r+q$;
reallocating A1, B5, C5 in *q* by *h*(*k*,*r*+1)
→ *q*++

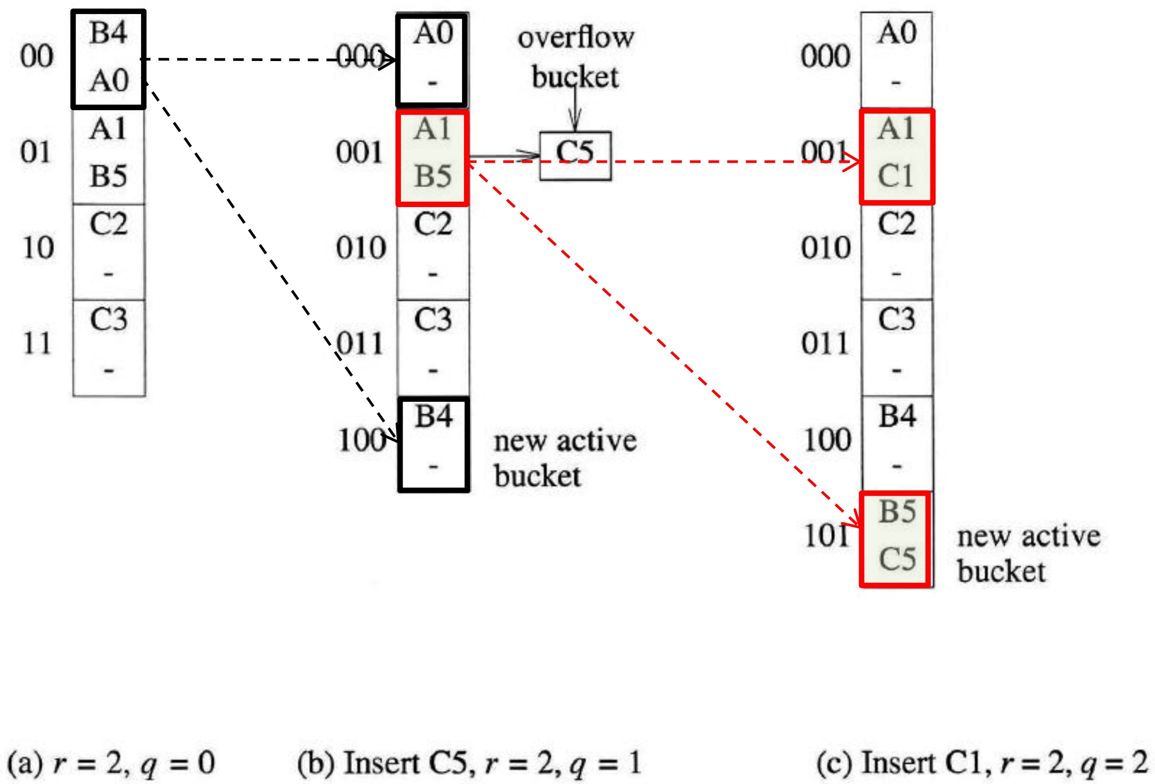
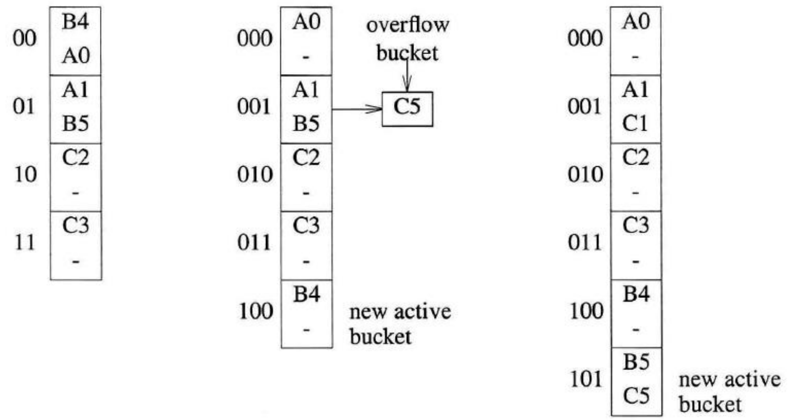



**Figure 8.9:** Inserting into a directoryless dynamic hash table

- Overflow
  - Handled by activating bucket $2^r+q$ , incrementing q by 1
  - In case q becomes $2^r$, increment r by 1 and reset q to 0




(a) $r = 2,\ q = 0$ (b) Insert C5, $r = 2,\ q = 1$ (c) Insert C1, $r = 2,\ q = 2$

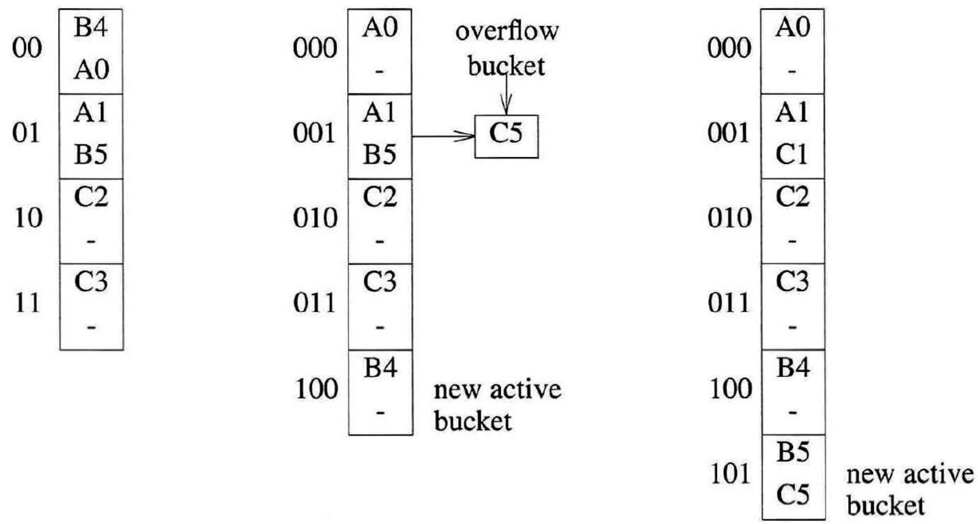
**Figure 8.9:** Inserting into a directoryless dynamic hash table

- Searchng

**if** ($h(k,r) < q$) search the chain that begins at bucket $h(k,r+1)$;
**else** search the chain that begins at bucket $h(k,r)$;

**Program 8.5:** Searching a directoryless hash table

| $k$ | $h(k)$ |
|---|---|
| A0 | 100 000 |
| A1 | 100 001 |
| B0 | 101 000 |
| B1 | 101 001 |
| C1 | 110 001 |
| C2 | 110 010 |
| C3 | 110 011 |
| C5 | 110 101 |




(a) $r = 2$, $q = 0$ (b) Insert C5, $r = 2$, $q = 1$ (c) Insert C1, $r = 2$, $q = 2$

# Hashing

- 8.1 Introduction
- 8.2 Static Hashing
  - Hash Tables
  - Hashing Functions
    - Mid-square, Division, Folding, Digit Analysis
  - Overflow Handling
    - Open addressing, Chaining
- 8.3 Dynamic Hashing