File Structures

Ch12. A. Extendible Hashing

2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

Slides are based on the textbook.

data science laboratory

Outline

- 12.1 Introduction
- 12.2 How extendible hashing works
- 12.3 Implementation

Introduction

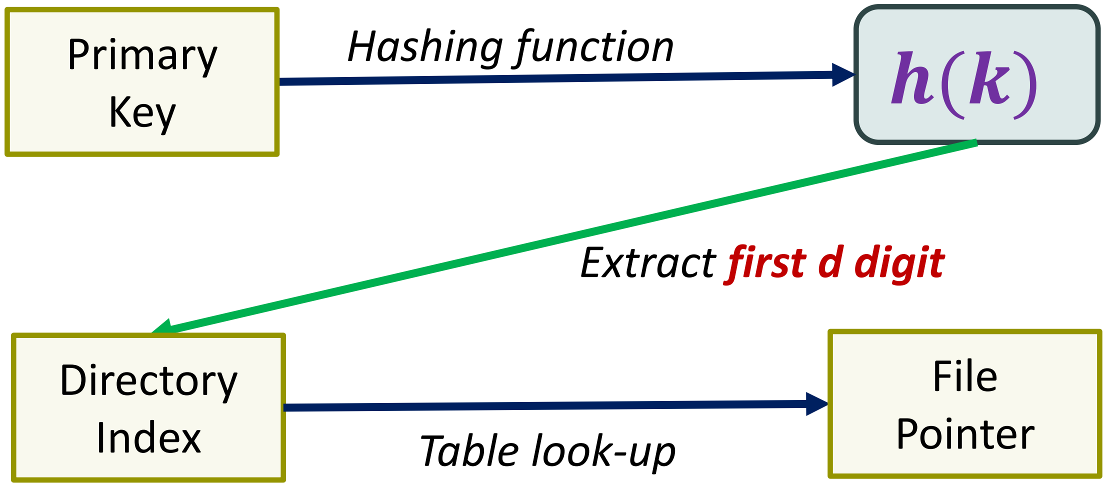
- Dynamic files
  - undergo a lot of growths
- Static hashing
  - described in chapter 11 (direct hashing)
  - typically worse than B-Tree for dynamic files
  - eventually requires file reorganization
- Extendible hashing
  - hashing for dynamic file
  - Fagin, Nievergelt, Pippenger, and Strong (ACM TODS 1979)

What is extendible hashing? (1/3)

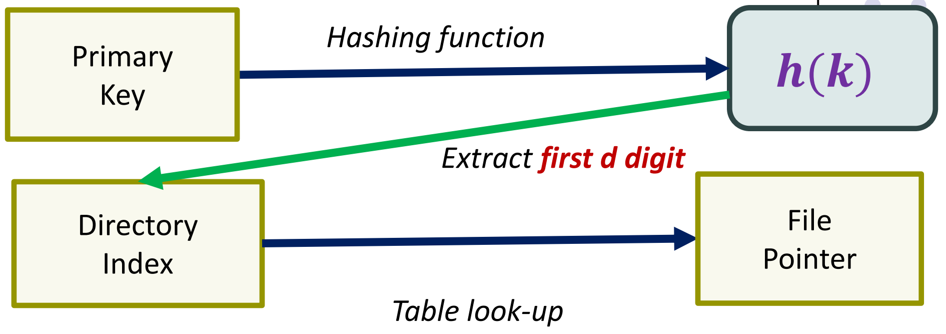
- Dynamic hashing approach
  - allows insertions and deletions to occur without resulting in poor performance after many of these operations
- Extendible hashing combines two ingredients:
  - hashing and tries.
  - (tries are digital trees like the one used in Lempel-Ziv)

What is extendible hashing? (2/3)

- Concept of Extendible Hashing



What is extendible hashing? (3/3)



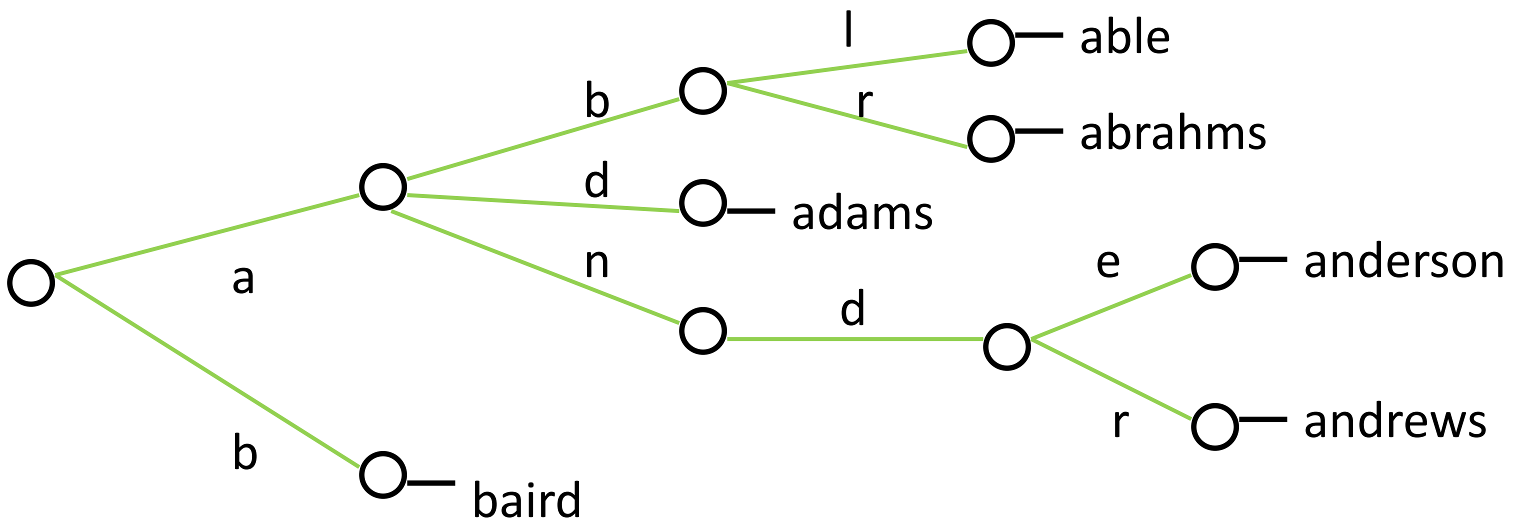
- Structures
  - Keys
    - are placed into buckets, which are independent parts of a file in disk.
    - having a hashing address with the same prefix share the same bucket.
  - A trie
    - is used for fast access to the buckets.
    - uses a prefix of the hashing address in order to locate the desired bucket

Outline

- 12.1 Introduction
- 12.2 How extendible hashing works
- 12.3 Implementation

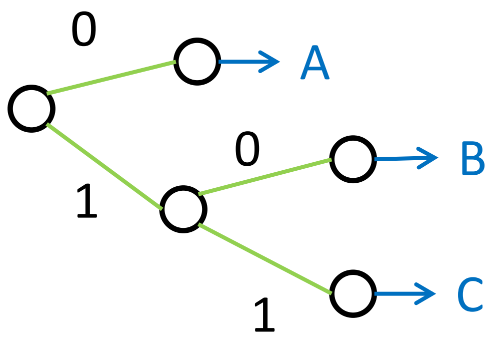
How Extendible Hashing works

- Idea from Tries file (radix searching)
  - The branching factor of the tree is equal to the # of alternative symbols in each position of the key
    - e.g.) Radix 26 trie - able, abrahms, adams, anderson, adnrews, baird
  - Use the first n characters for branching



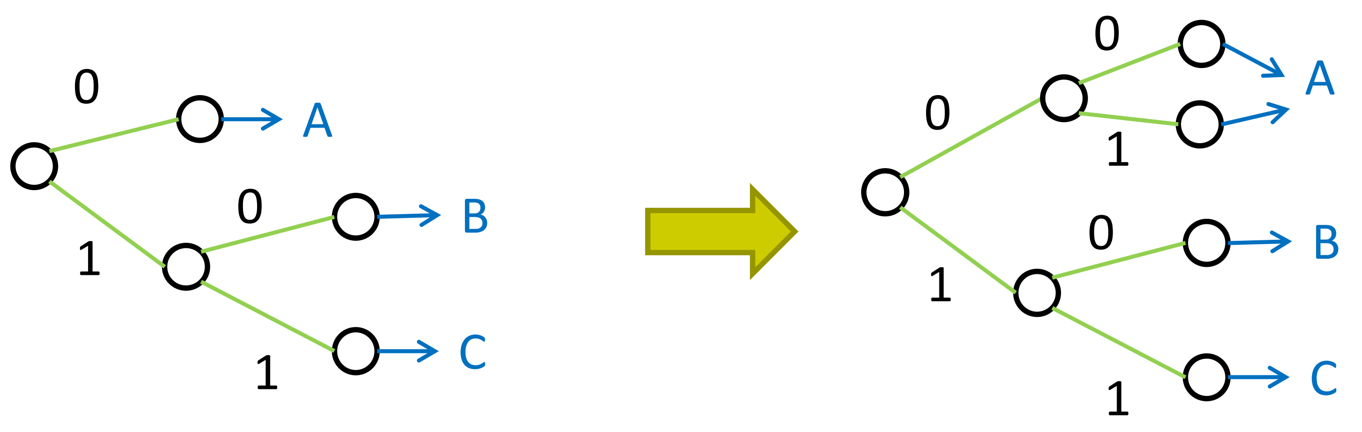
Tries and buckets

- Consider the following grouping of keys into buckets, depending on the prefix of their hash addresses:
  - bucket: this bucket contains keys with hash address with prefix:
    - A 0
    - B 10
    - C 11



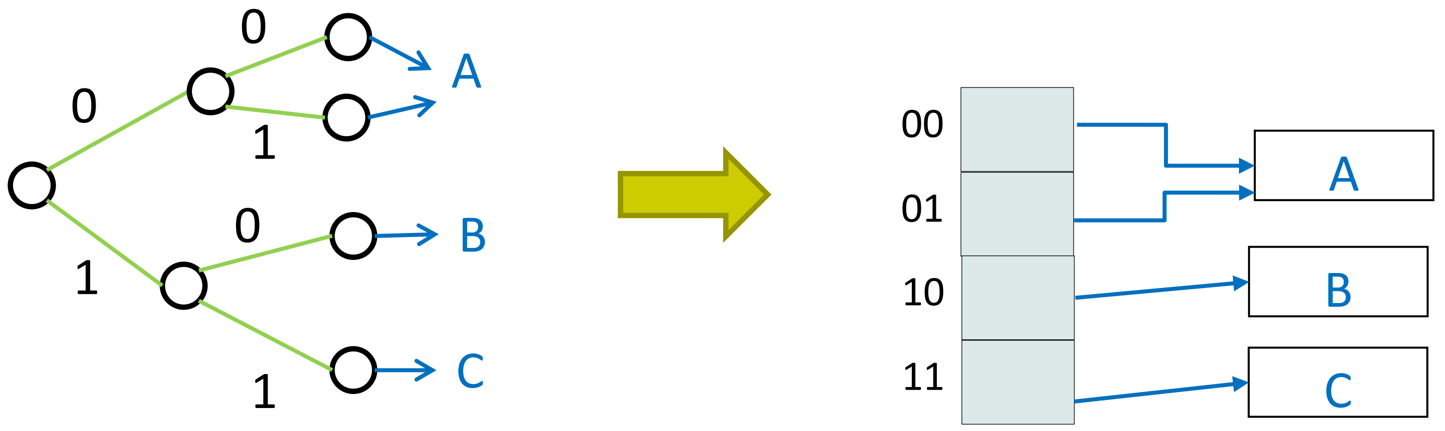
Directory structure and buckets (1/4)

- Representing the trie as a tree would take too much space.
- Instead, do the following:
  - 1) Extend the tree to a complete binary tree:



Directory structure and buckets (2/4)

- Instead, do the following:
  - 2) Flatten the trie up into an array:

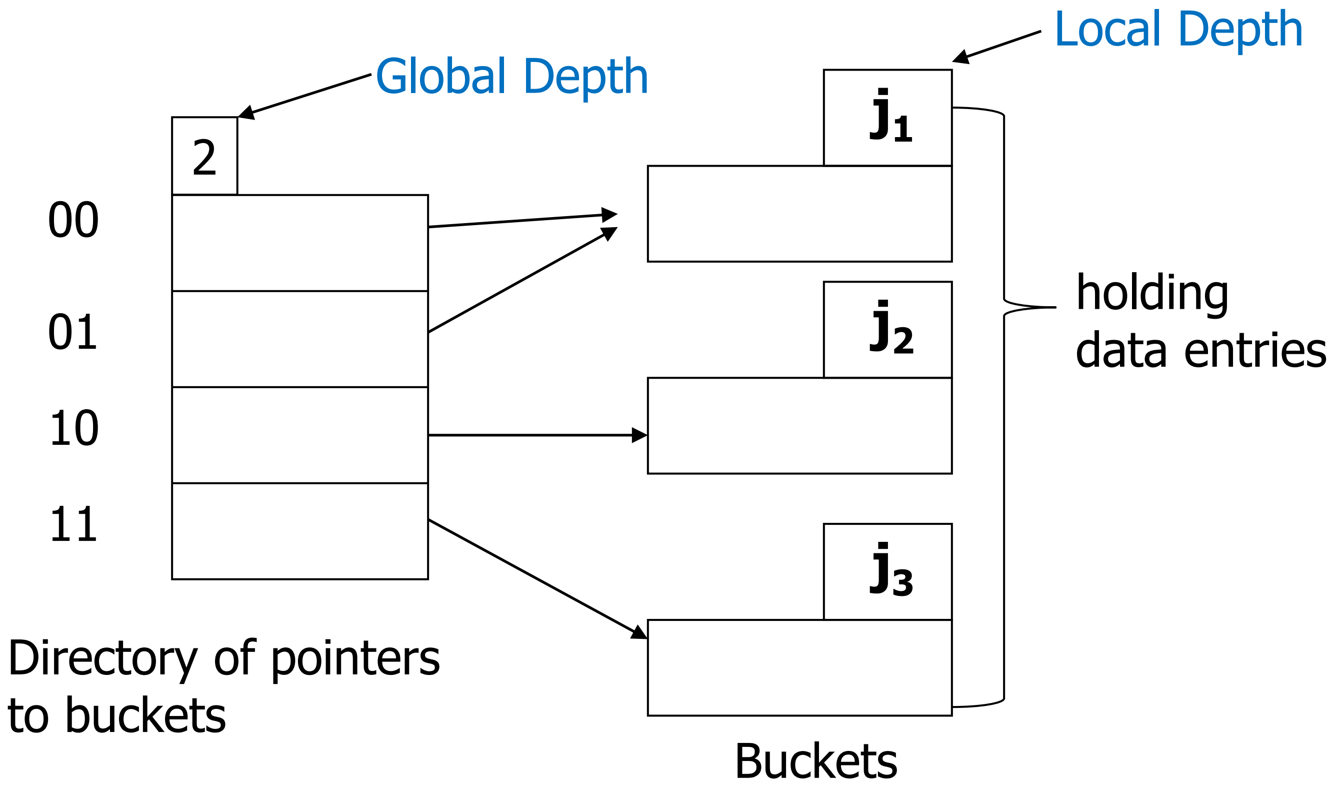


Directory structure and buckets (3/4)

- Directory is a complete binary tree
  - Directory entry : a pointer to the associated bucket
  - Given an address beginning with the bits 10, the 2^{10} directory entries
  - Introduced for uniform distribution

Directory structure and buckets (4/4)

- Global depth of directory (i)
  - First i bits of a binary number to tell which bucket an entry belongs to
- Local depth of bucket (j)
  - Hash values of data entries in bucket agree on the first j bits



Searching for a record

- Steps in retrieving a record with a given key
  - find H(given key)
  - extract first d bits of H(given key)
  - use this value as an index into the directory to find a pointer
  - use this pointer to read a bucket into primary memory
  - locate the desired record within the bucket (scan)

What makes it extendible?

- Making the hash table expand or shrink as records are added or deleted.

- The dynamic aspects
  - Insertions and bucket splitting.
  - Deletions and bucket combination.

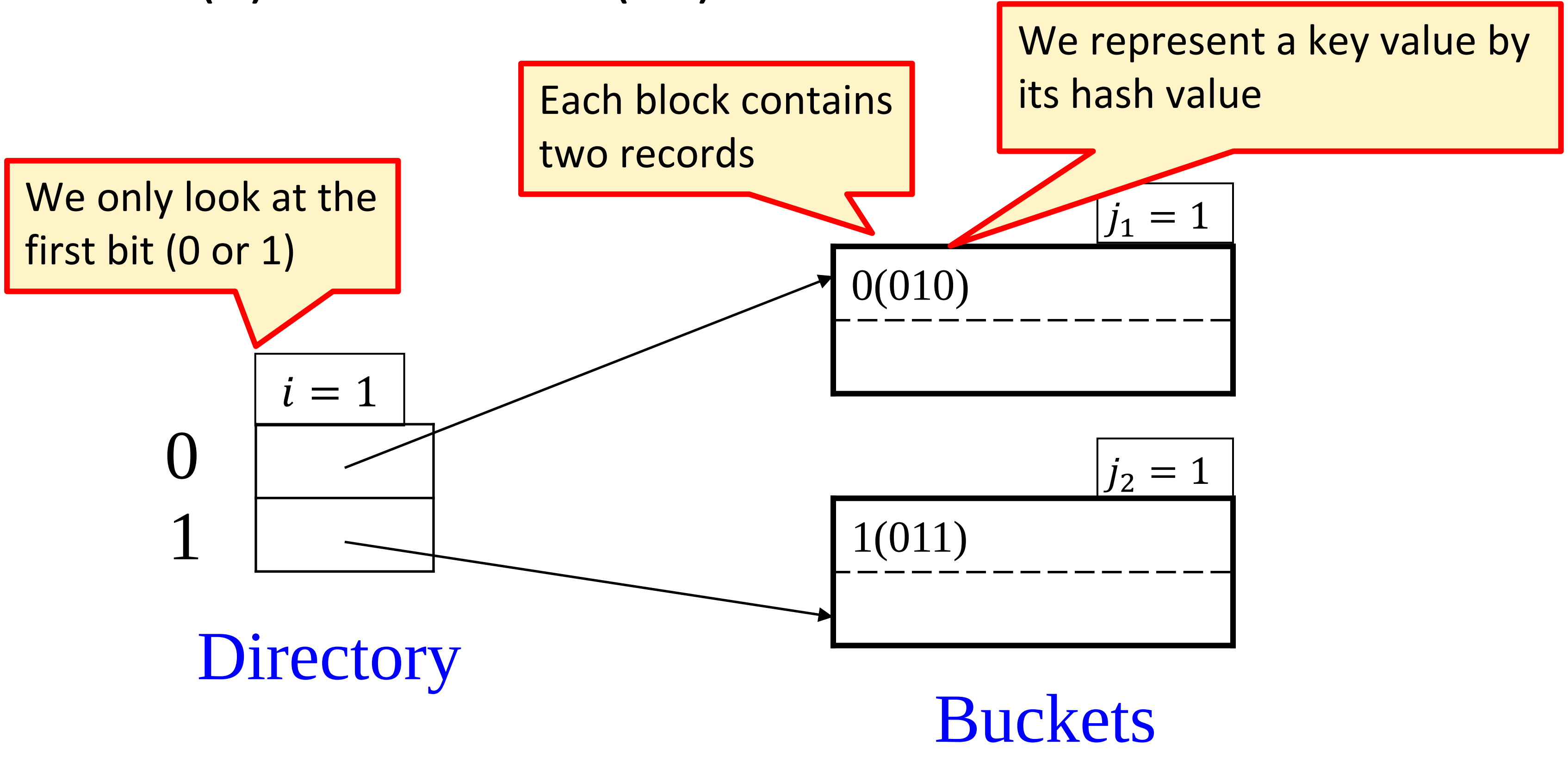
Bucket splitting to handle overflow

- Extendible hashing solves bucket overflow
  - The directory's size is a power of 2 and may vary;
    - double # of buckets by doubling the directory
    - splitting just the bucket that overflowed!

- Let us assume the keys are numbers and the hash function returns the number itself.
  - For instance, h(20) = 20.
  - Bucket size: 2

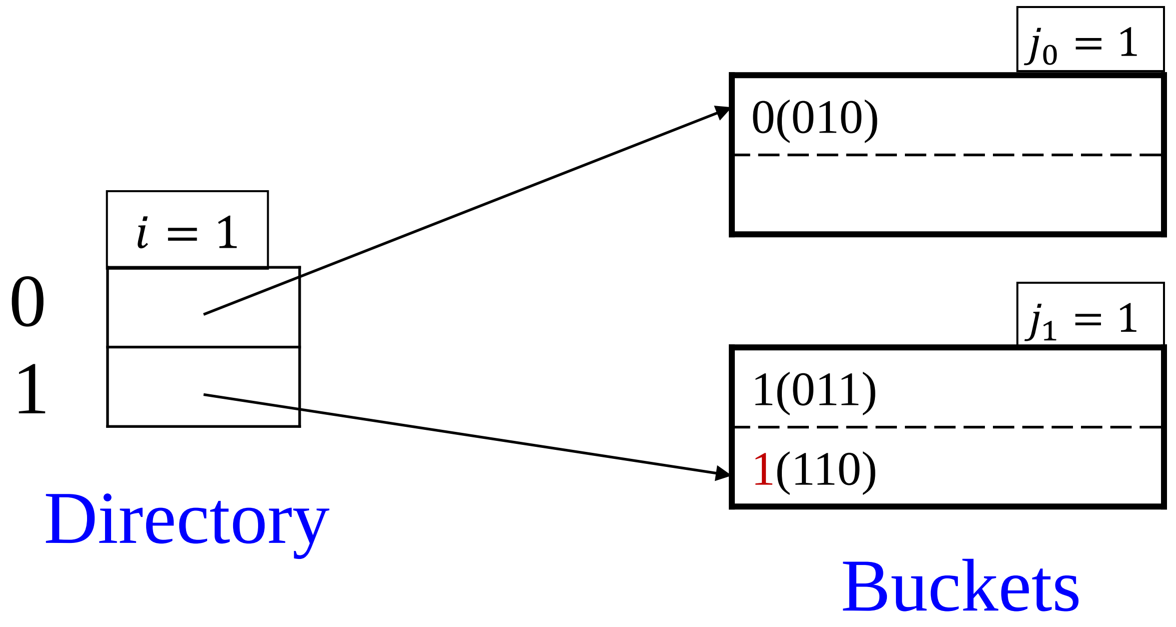
Insertion in Extensible Hash Table(1/9)

- 1. Insert 2 (0010) and 11 (1011)
  - H(2)=0010 and H(11)= 1011


We only look at the first bit (0 or 1)
Each block contains two records
We represent a key value by its hash value
$i = 1$
0
1
$j_1 = 1$
0(010)
$j_2 = 1$
1(011)
Directory
Buckets

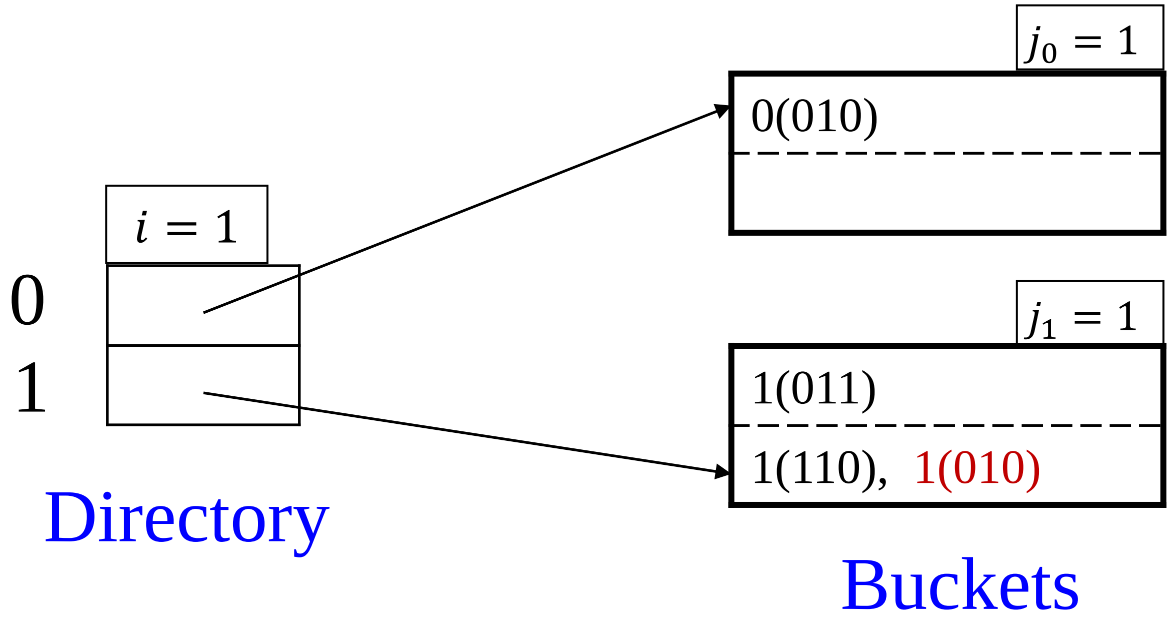
Insertion in Extensible Hash Table(2/9)

- 2. Insert 14 (1110)
  - h(14)=1110



Insertion in Extensible Hash Table(3/9)

- 3. insert 10 (1010)



  - Need to extend table, split blocks
  - Global i becomes 2

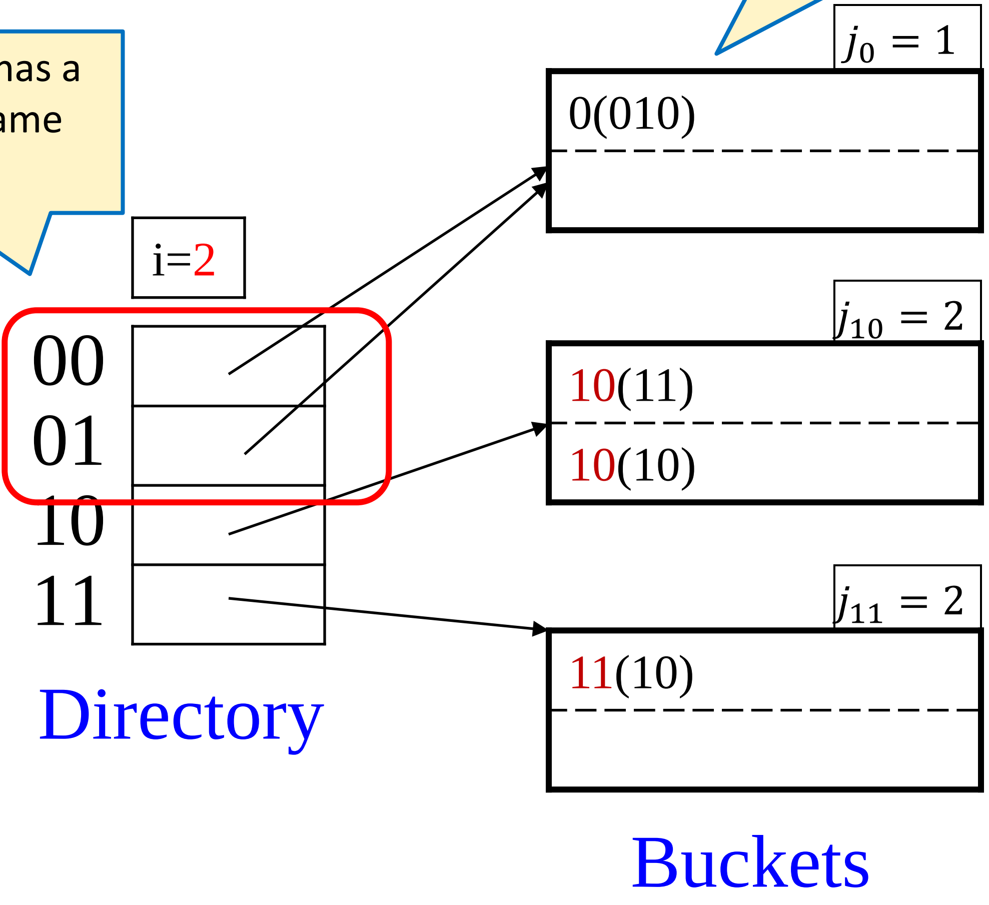
Insertion in Extensible Hash Table(4/9)

- 3. insert 10 (1010) (cont.)

We don’t need to split this block yet.

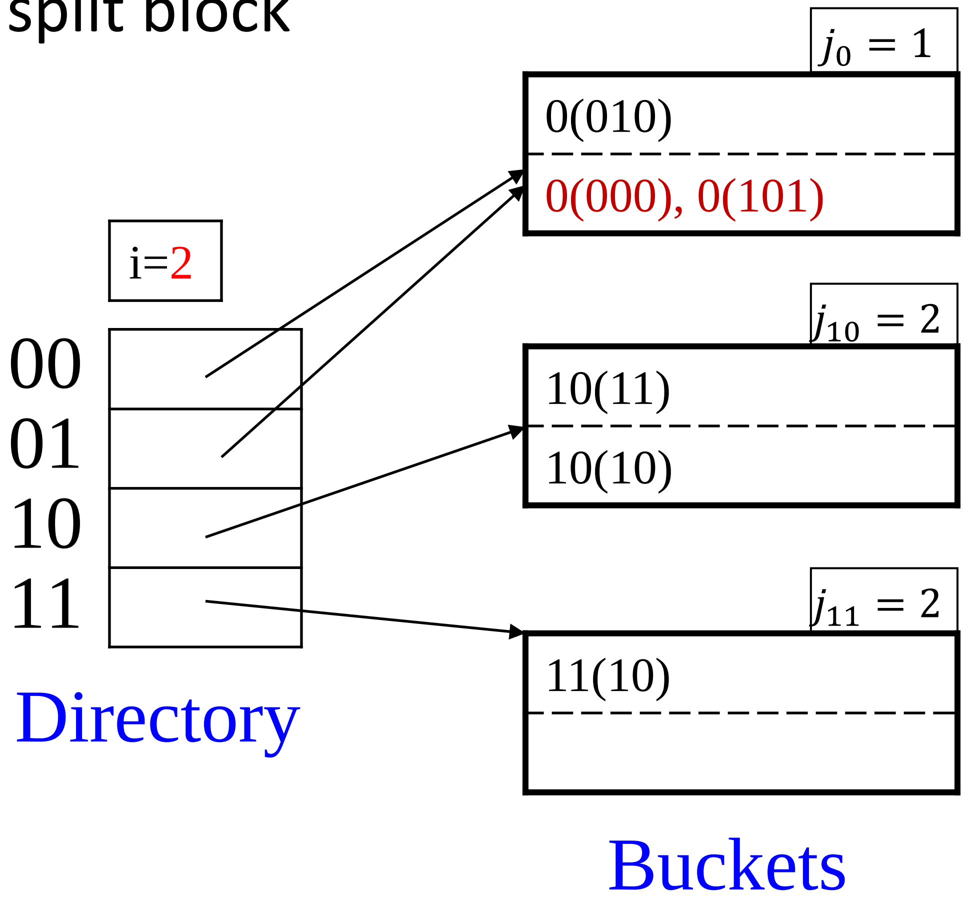
The first entries has a pointer to the same block.

The first 2 bits are considered to decide the membership for this block.



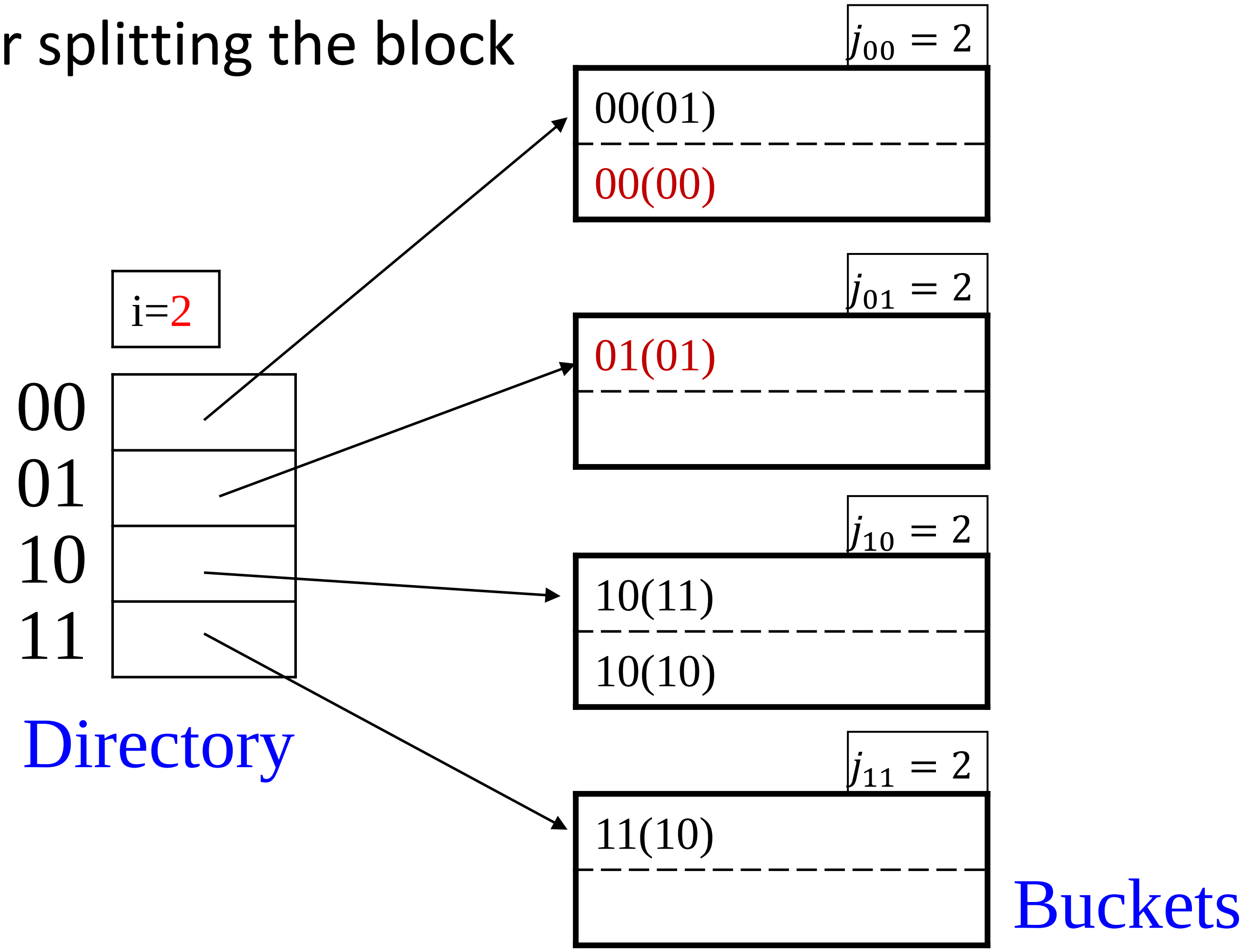
Insertion in Extensible Hash Table(5/9)

- 4. insert 0 (0000), then 5 (0101)
  - Need to split block



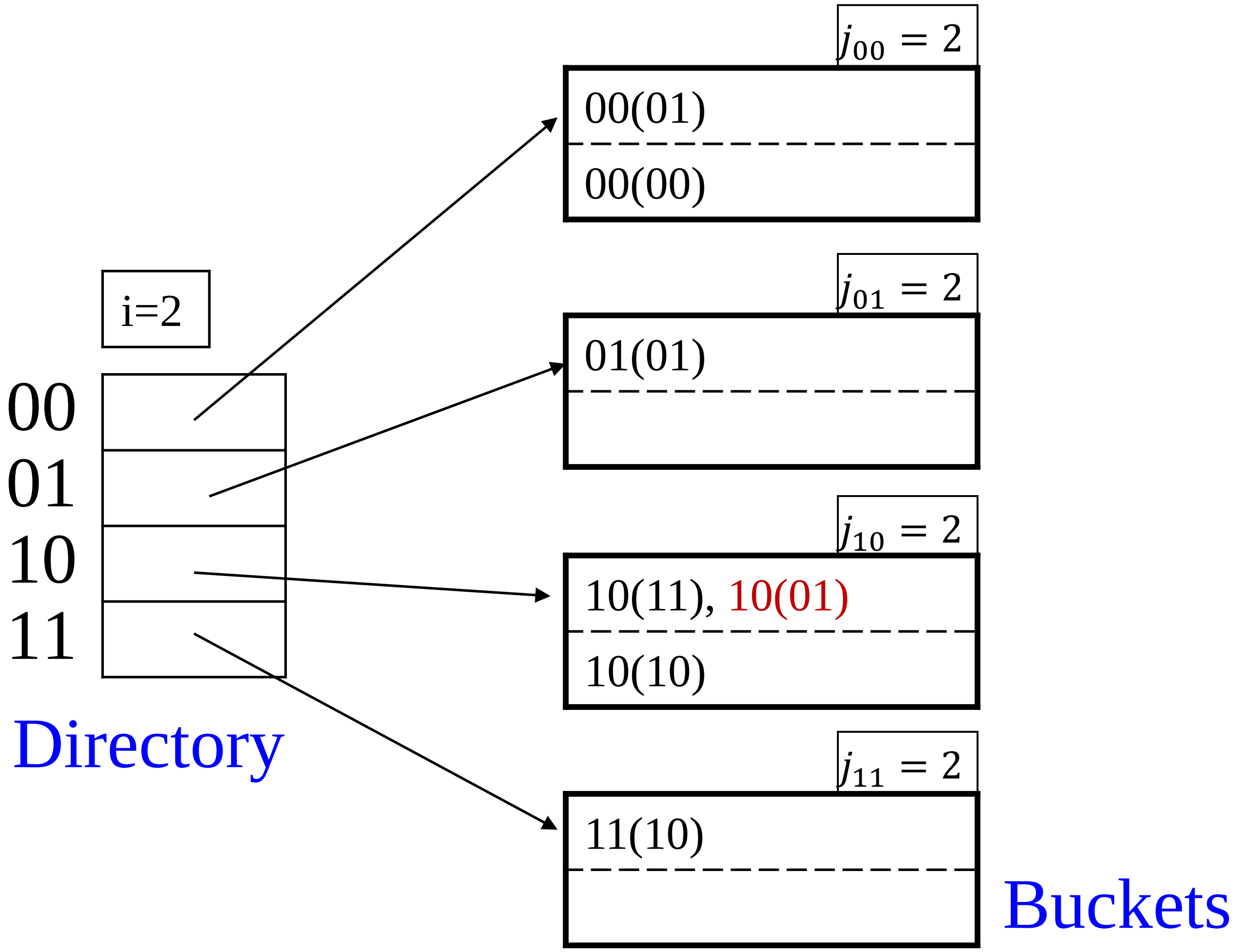
Insertion in Extensible Hash Table(6/9)

- 4. insert 0 (0000), then 5 (0101) (cont.)
  - After splitting the block



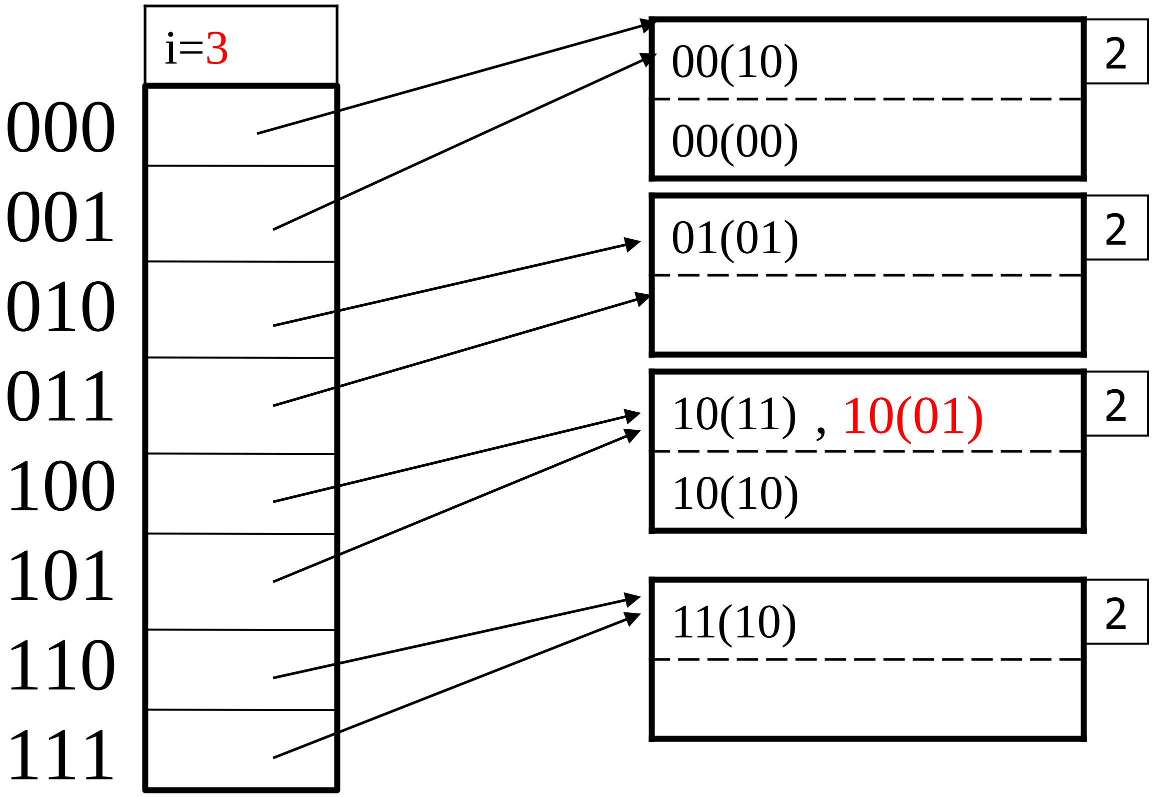
Insertion in Extensible Hash Table(7/9)

- 5. Insert 9 (1001)



Insertion in Extensible Hash Table(8/9)

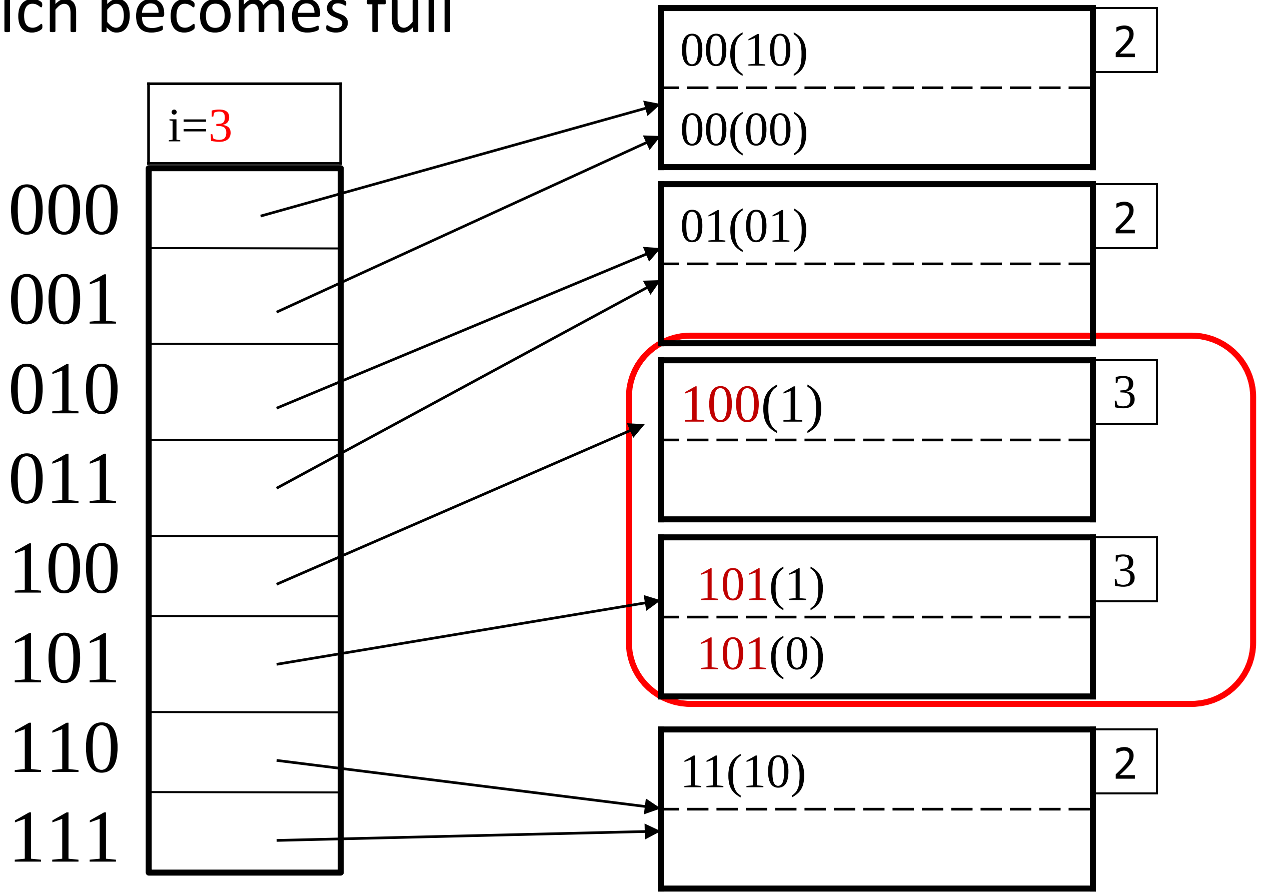
- 5. Insert 9 (1001) (cont'd)
  - Increment i by 1, i.e., double the length of an array



Each two entries share the same block

Insertion in Extensible Hash Table(9/9)

- 5. Insert 9 (1001) (cont'd)
  - Create a new block and distribute records in the block, which becomes full



Insertion Algorithm

- Calculate the hash function for the key and the key address in the current directory.
- Follow the directory address to find the bucket that should receive the key.
- Insert into the bucket, splitting it if necessary.
- If splitting took place
    - calculate i: the number of bits necessary to differentiate keys within this bucket
    - Double the directory as many times as needed to create a directory indexed by i bits.

Outline

- 12.1 Introduction
- 12.2 How extendible hashing works
- 12.3 Implementation

Creating the address (1/5)

- Function hash(KEY)
  - Fold/Add hashing algorithm
  - Do not MOD hashing value by address space since no fixed address space exists
  - Output from the hash function for a number of keys

```
bill            0000 0011 0110 1100
lee             0000 0100 0010 1000
pauline         0000 1111 0110 0101
alan            0100 1100 1010 0010
julie           0010 1110 0000 1001
mike            0000 0111 0100 1101
elizabeth       0010 1100 0110 1010
mark            0000 1010 0000 0111
```

Distribution of the least significant bits of these integers tends to have more variation than the high-order bits

Creating the address (2/5)

- Hash()
  - Return an integer hash value for key for a 15-bit
    - Modulo 19937: the character summation stays within the range of a signed 16-bit integer

```
int Hash (char* key)
{
    int sum = 0;
    int len = strlen(key);
    if (len % 2 == 1) len ++; // make len even
    // for an odd length string,
    // use the trailing '\0' as part of key
    for (int j = 0; j < len; j += 2)
        sum = (sum + 100 * key[j] + key[j+1]) % 19937;
    return sum;
}
```

Creating the address (3/5)

- MakeAddress()
  - extracts just a portion of the full hashed address
  - **reverse the order of the bits of the value**
  - A parameter depth:
    - The number of address bits to return

```
int MakeAddress (char * key, int depth)
{
    int retval = 0;
    int mask = 1;
    int hashVal = Hash(key);
    // reverse the bits
    for (int j = 0; j < depth; j++)
    {
        retval = retval << 1;
        int lowbit = hashVal & mask;
        retval = retval | lowbit;
        hashVal = hashVal >> 1;
    }
    return retval;
}
```

Creating the address (4/5)

```
for (int j = 0; j < depth; j++)
{
        retval = retval << 1;
        int lowbit = hashVal & mask;
        retval = retval | lowbit;
        hashVal = hashVal >> 1;
}
```

- Example
  - `hashVal = Hash(pauline);`
    - `hashVal: 0000 1111 0110 0101`
  - Assume: depth=16

hashVal	00001111 01100101

j=0

retval <<1	0000 0000 0000 0000
Lowbit = hashVal & mask (0x1)	0000 0000 0000 0001
retval = retval | lowbit	0000 0000 0000 0001
hashVal >> 1	**0000 0111 1011 0010**

j=1

retval <<1	0000 0000 0000 0010
Lowbit = hashVal & mask (0x1)	0000 0000 0000 0000
retval = retval | lowbit	0000 0000 0000 0010
hashVal >> 1	**0000 0011 1101 1001**

Creating the address (5/5)

```
for (int j = 0; j < depth; j++)
{
    retval = retval << 1;
    int lowbit = hashVal & mask;
    retval = retval | lowbit;
    hashVal = hashVal >> 1;
}
```

- Example

hashVal	0000 1111 0110 0101

j=2

retval <<1	0000 0000 0000 0100
Lowbit = hashVal & mask (0x1)	0000 0000 0000 0001
retval = retval | lowbit	0000 0000 0000 0101
hashVal >> 1	0000 0001 1110 1100

j=3

retval <<1	0000 0000 0000 1010
Lowbit = hashVal & mask (0x1)	0000 0000 0000 0000
retval = retval | lowbit	0000 0000 0000 1010
hashVal >> 1	0000 0000 1111 0110

hashVal	0000 1111 0110 0101

$\longleftarrow$

retVal	1010 0110 1111 0000

Extensible hashing Scheme

- Two main components
  - (1) A set of buckets stored in a file
    - Store a key-reference pair in a bucket
    - The set of buckets is similar to the sequence set of B+-tree
      - each bucket contains a set of keys and information associated with the keys
    - Forms an index of the file of actual records
  - (2) A directory
    - an array containing the record addresses of the buckets

Class for Bucket Objects (1/2)

- The basic operations on buckets
  - Exactly the same as those of index record
    - Class Bucket is a derived class of the class TextIndex
    - Bucket records are stored in a file
    - Each Bucket is connected to a directory
      - Protected: no outside access is allowed
  - three operations
    - add a key-reference pair to a bucket
    - Search for a key and return its reference
    - Remove a key

Class for Bucket Objects (2/2)

```
class Bucket: public TextIndex
{
protected:
    // there are no public members,
    // access to Bucket members is only through class Directory
    Bucket (Directory & dir, int maxKeys = defaultMaxKeys); // constructor
    int Insert (char * key, int recAddr);
    int Remove (char * key);
    Bucket * Split ();// split the bucket and redistribute the keys
    // calculate the range of a new (split) bucket
    int NewRange (int & newStart, int & newEnd);
    int Redistribute (Bucket & newBucket); // redistribute keys
    int FindBuddy ();// find the bucket that is the buddy of this
    int TryCombine (); // attempt to combine buckets
    int Combine (Bucket * buddy, int buddyIndex); // combine two buckets
    // number of bits used 'in common'
    // by the keys in this bucket
    int Depth;
    Directory & Dir; // directory that contains the bucket
    int BucketAddr; // address of file
    ostream & Print (ostream &);
    friend class Directory;
    friend class BucketBuffer;
};
```

Class for Directory Objects

- Class Directory
  - Each cell in the directory consists of the file address of a bucket record
  - use direct access to find directory cells
    - Implement the directory as an array of directory cells in memory
      - treat the address values returned by MakeAddress() as subscripts for this array
  - Additional member functions
    - To support the file operations required store the directory and buckets
      - A striking similarity with IndexedFile and Btree class

Detour: B-Tree

```
template <class keyType>
class BTree
{
public:
    BTree(int order, int keySize = sizeof(keyType), int unique = 1);
    ~BTree();
    int Open (char * name, ios_base::openmode mode);
    int Create (char * name, ios_base::openmode mode);
    int Close ();
    int Insert (const keyType key, const int recAddr);
    int Remove (const keyType key, const int recAddr = -1);
    int Search (const keyType key, const int recAddr = -1);
    void Print (ostream &);
    void Print (ostream &, int nodeAddr, int level);
protected:
    typedef BTreeNode<keyType> BTNode;// useful shorthand
    BTNode * FindLeaf (const keyType key);
    …
    BTNode ** Nodes; // pool of available nodes
    // Nodes[1] is level 1, etc. (see FindLeaf)
    // Nodes[Height-1] is leaf
    FixedFieldBuffer Buffer;
    RecordFile<BTNode> BTreeFile;
};
```

Directory class (1/2)

```
class Directory
{
public:
    Directory (int maxBucketKeys = -1);
    ~Directory ();
    int Open (char * name);
    int Create (char * name);
    int Close ();
    int Insert (char * key, int recAddr);
    int Remove (char * key);
    int Search (char * key); // return RecAddr for key
    ostream & Print (ostream & stream);
protected:
    int Depth; // depth of directory
    int NumCells; // number of entries, = 2**Depth
    int * BucketAddr; // array of bucket addresses

    // protected methods
    int DoubleSize (); // double the size of the directory
    int Collapse (); // collapse, halve the size
    int InsertBucket (int bucketAddr, int first, int last);
    int RemoveBucket (int bucketIndex, int depth);// remove bucket from directory
    int Find (char * key); // return BucketAddr for key
    int StoreBucket (Bucket * bucket);// update or append bucket in file
    int LoadBucket (Bucket * bucket, int bucketAddr);// load bucket from file
```

Directory class (2/2)

```
    // methods to support Remove

    // members to support directory and bucket files
    int MaxBucketKeys;
    BufferFile * DirectoryFile;
    LengthFieldBuffer * DirectoryBuffer;
    Bucket * CurrentBucket;// object to hold one bucket
    BucketBuffer * theBucketBuffer;// buffer for buckets
    BufferFile * BucketFile;
    int Pack () const;
    int Unpack ();
    Bucket * PrintBucket;// object to hold one bucket

    friend class Bucket;
};
```

- Use objects of class BufferFile to provide the I/O operations

Test program (1/2)

```
main ()
{
    int result;
    Directory Dir (4);
    result = Dir . Create ("tsthash");
    if (result == 0)
    {
        cout<<"Please delete tsthash.dir and tsthash.bkt"<<endl;
        return 0;
    }
    char * keys[]={"bill", "lee", "pauline", "alan", "julie",
        "mike", "elizabeth", "mark", "ashley", "peter",
        "joan", "john", "charles", "mary", "emily"};
    const int numkeys = 15;
    for (int i = 0; i<numkeys; i++)
    {
        cout <<keys[i]<<" "<<(void*)Hash(keys[i])
            <<" "<<(void*)MakeAddress(keys[i],16)<<endl;
        result = Dir . Insert (keys[i], 100 + i);
        if (result == 0) cout << "insert for "<<keys[i]<<" failed"<<endl;
        Dir . Print (cout);
    }
    return 1;
}
```

Test program (2/2)

- To use of Directory object
  - Must be constructed and then attached to a file for directory and one for the bucket
- The two steps for initialization
  - 1) The declaration of the Directory object
  - 2) The call to method Create
    - Create the two files and the initial empty bucket

Directory::Directory()

- The constructor of Directory
  - creates all of the objects that support the I/O operation
    - A buffer and a file for the directory
    - A buffer and a file for the buckets

```
Directory::Directory (int maxBucketKeys)
{
    Depth = 0; // depth of directory
    NumCells = 1; // number of entries, = 2**Depth
    BucketAddr = new int [NumCells]; // array of bucket addresses

    MaxBucketKeys = maxBucketKeys;
    DirectoryBuffer = new LengthFieldBuffer; // default size
    DirectoryFile = new BufferFile(*DirectoryBuffer);
    CurrentBucket = new Bucket (*this, MaxBucketKeys);
    theBucketBuffer = new BucketBuffer (MaxKeySize, MaxBucketKeys);
    BucketFile = new BufferFile (*theBucketBuffer);
    PrintBucket = new Bucket (*this, MaxBucketKeys);
}
```

Directory::Create (1/2)

```
int Directory::Create (char * name)
{
    // create the two files, clear the directory, create a single bucket
    // and add it to the directory and the bucket file
    int result;
    char * directoryName, * bucketName;
    makeNames(name, directoryName, bucketName);
    result = DirectoryFile->Create(directoryName, ios::in|ios::out);
    if (!result) return 0;
    result = BucketFile->Create(bucketName,ios::in|ios::out);
    if (!result) return 0;
    // store the empty CurrentBucket in the BucketFile and add to Directory
    BucketAddr[0] = StoreBucket (CurrentBucket);
    return result;
}
```

- The directory is stored in memory while the directory is open

Directory::Create (2/2)

- Size of the directory
  - 2^{depth} = the number of cells in the directory
- Starting a new hash directory
  - Directory depth is 0
  - Using no bits to distinguish between address
  - All keys go into the same bucket
  - use number bits to distinguish between addresses
- get the address of the initial, everything-goes-here bucket

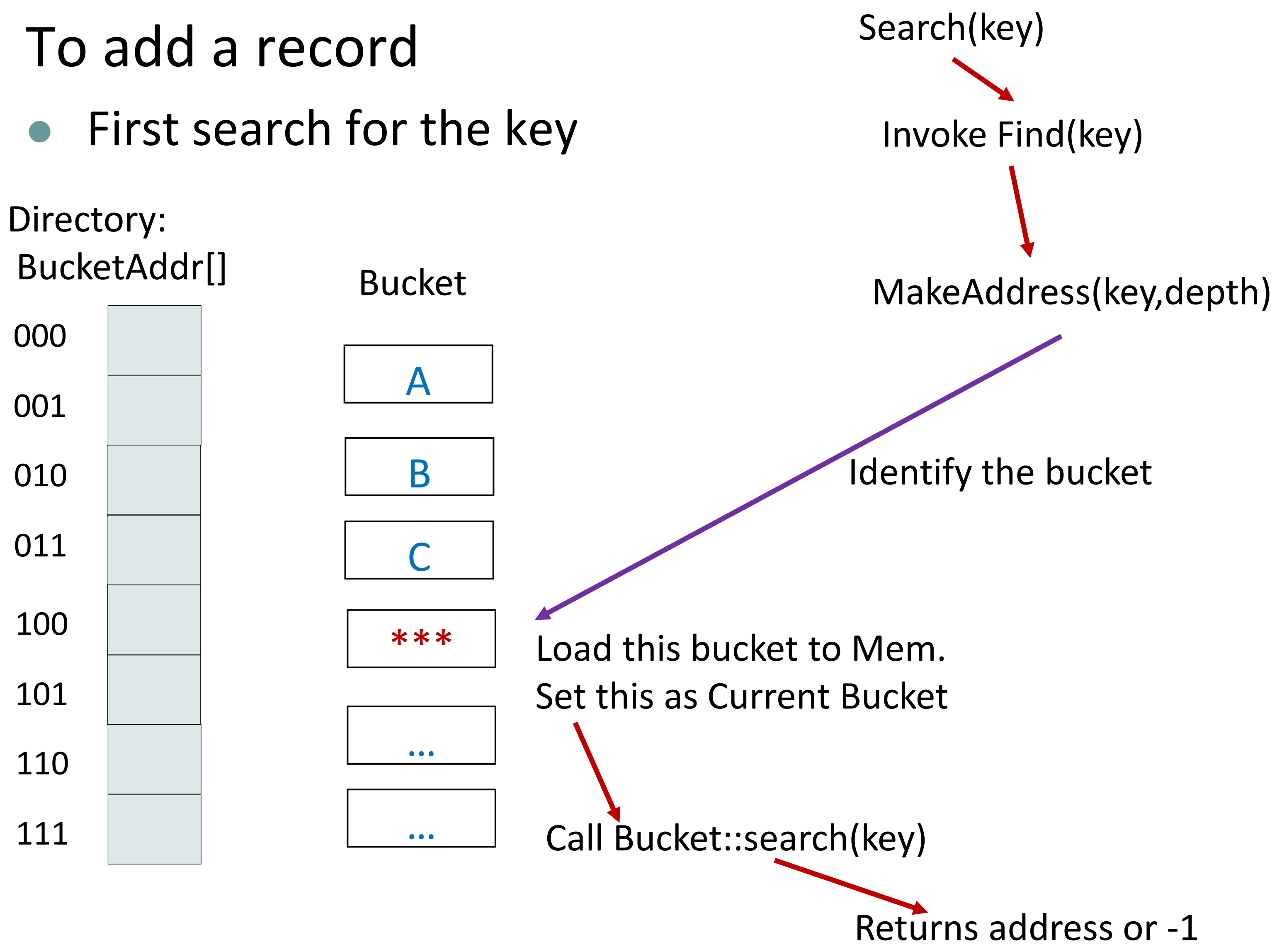
```
BucketAddr[0] = StoreBucket (CurrentBucket);
```

Outline

- 12.1 Introduction
- 12.2 How extendible hashing works
- 12.3 Implementation
  - 12.3.3 Directory and bucket operations

Add records to Directory (1/4)

- To add a record
  - First search for the key



Add records to Directory (2/4)

- Insert
  - Searches for the key
    - Case1: found -> done
    - Case2: not found
      - The key is added into the current bucket
      - Note that CurrentBucket is identified by Search()
      - call Bucket::Insert()

```
int Directory::Insert (char * key, int recAddr)
{
    int found = Search (key);
    if (found == -1) return CurrentBucket->Insert(key, recAddr);
    return 0;// key already in directory
}
```

Add records to Directory (3/4)

- Search
  - Arranges for the CurrentBucket to contain the proper bucket for the key
- Find
  - Determines which bucket is associate wit the key

```
// return RecAddr for key, also put current bucket into variable
int Directory::Search (char * key)
{
    int bucketAddr = Find(key);
    LoadBucket (CurrentBucket, bucketAddr);
    return CurrentBucket->Search(key);
}

// return BucketAddr for key
int Directory::Find (char * key)
{
    return BucketAddr[MakeAddress (key, Depth)];
}
```

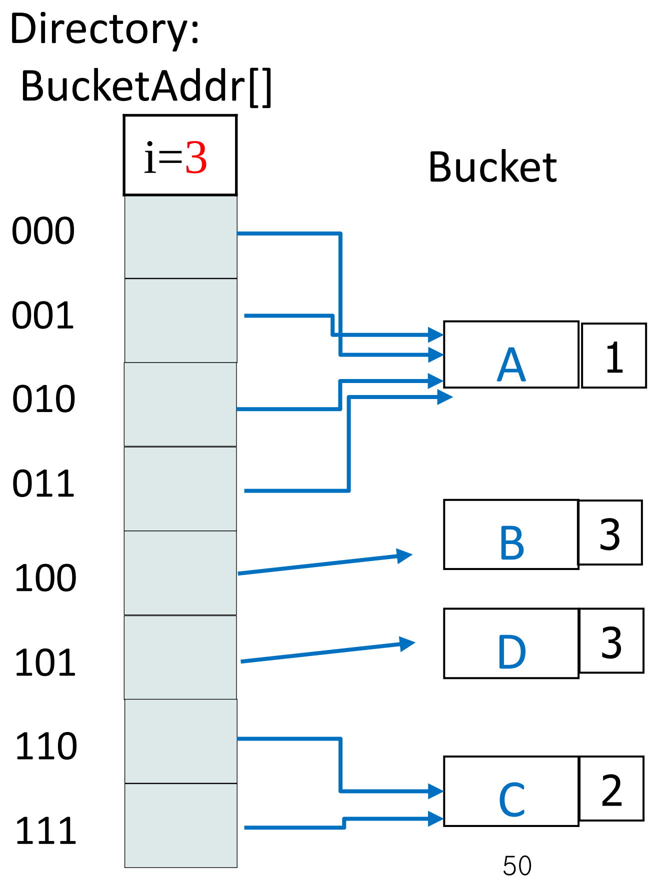
Add records to Directory (4/4)

- Bucket::Insert
  - Case1: Bucket is not full
    - Calls TextIndex::Insert() (key-reference pair)
  - Cae2: Full Bucket
    - Bucket split()
    - Recursively invokes Directory::Insert() to try again the insertion

```
int Bucket::Insert (char * key, int recAddr)
{
    if (NumKeys < MaxKeys){
        int result = TextIndex::Insert (key, recAddr);
        Dir.StoreBucket (this);
        return result;
    }
    else {
        Split ();
        return Dir.Insert (key, recAddr);
    }
}
```

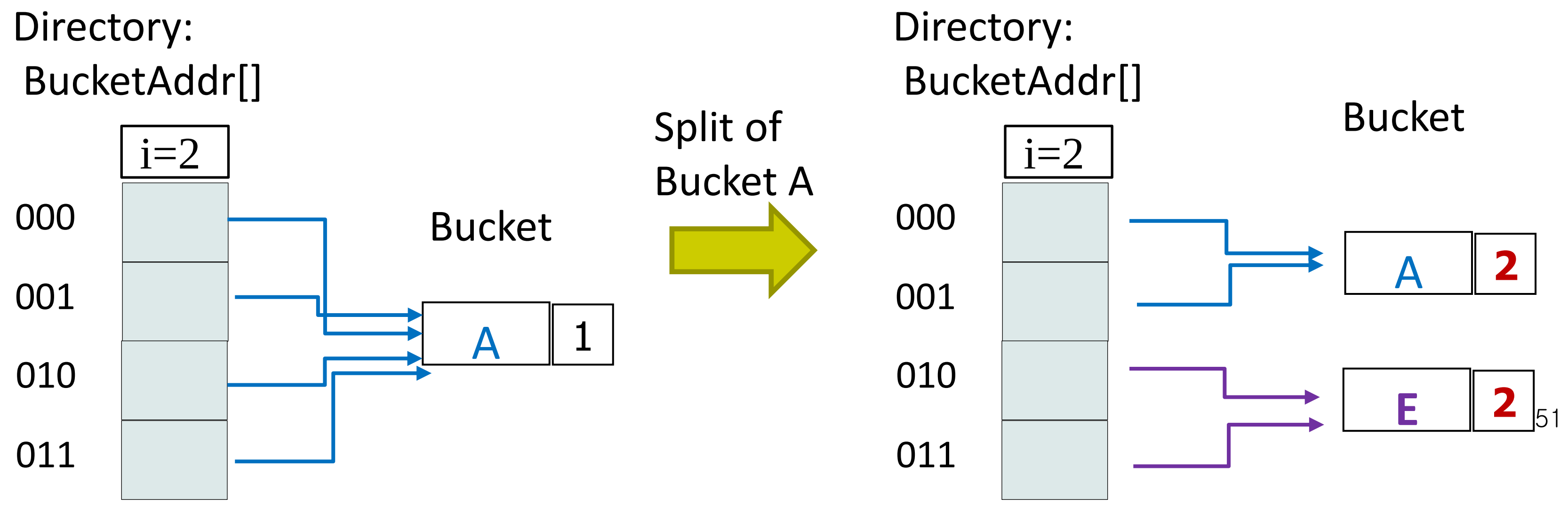
Bucket::Split() (1/5)

- An implementation of Split()
  - Consider the relationship between the global depth and the local depth
    - Directory: global depth=3
    - Bucket A
      - Local depth =1
    - Bucket C
      - Local depth=2
      - all share a common first 2 bits
    - Buckets B and D
      - Local depth=3



Bucket::Split() (2/5)

- Case 1
  - Split one of the buckets that is using fewer address bits than the directory
    - Referenced more than one directory cell
    - half of the directory cells to point to the new bucket after split
    - Do not have to expand the directory



Bucket::Split() (3/5)

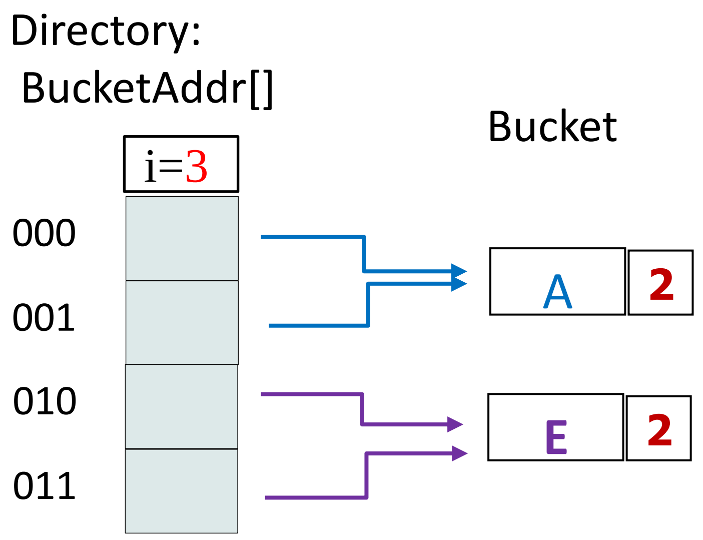
- Case 2
  - Split a bucket that has the same address depth as the directory
    - No additional directory cells to reference the new bucket
    - Before split the bucket → double the size of the direcotry

Bucket::Split() (4/5)

```
Bucket * Bucket::Split ()
{
    int newStart, newEnd;
    if (Depth == Dir.Depth)// no room to split this bucket
        Dir.DoubleSize();
    Bucket * newBucket = new Bucket (Dir, MaxKeys);
    Dir.StoreBucket (newBucket);
    NewRange (newStart, newEnd);
    Dir.InsertBucket(newBucket->BucketAddr, newStart, newEnd);
    Depth ++;
    newBucket->Depth = Depth;
    Redistribute (*newBucket);
    Dir.StoreBucket (this);
    Dir.StoreBucket (newBucket);
    return newBucket;
}
```

Bucket::Split() (5/5)

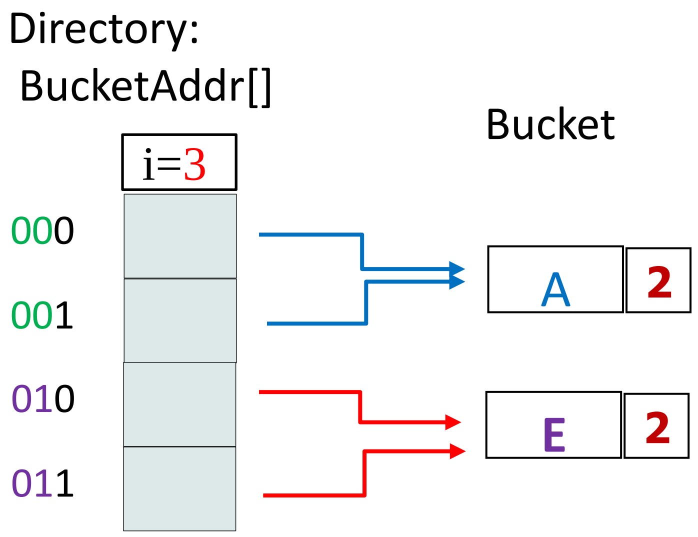
- Implementation
  - Compare the global depth and the local depth
    - If needed, double the directory size
  - Create the new bucket that we need for the split
    - Find the range of directory addresses to be used for the new bucket
    - Example: Split of bucket A as A and E
      - The range of directory addresses for the new bucket : 010 – 011



Bucket::NewRange (1/3)

- find the range of directory cells that should point to the new bucket instead of the old one after the split

- Example1: Split of bucket A
  - Bucket A share an 00
  - New E share an 01
  - Directory uses 3 bits
    - the new bucket is attached to the directory cells starting with 010 and ending with 011



Bucket::NewRange (2/3)

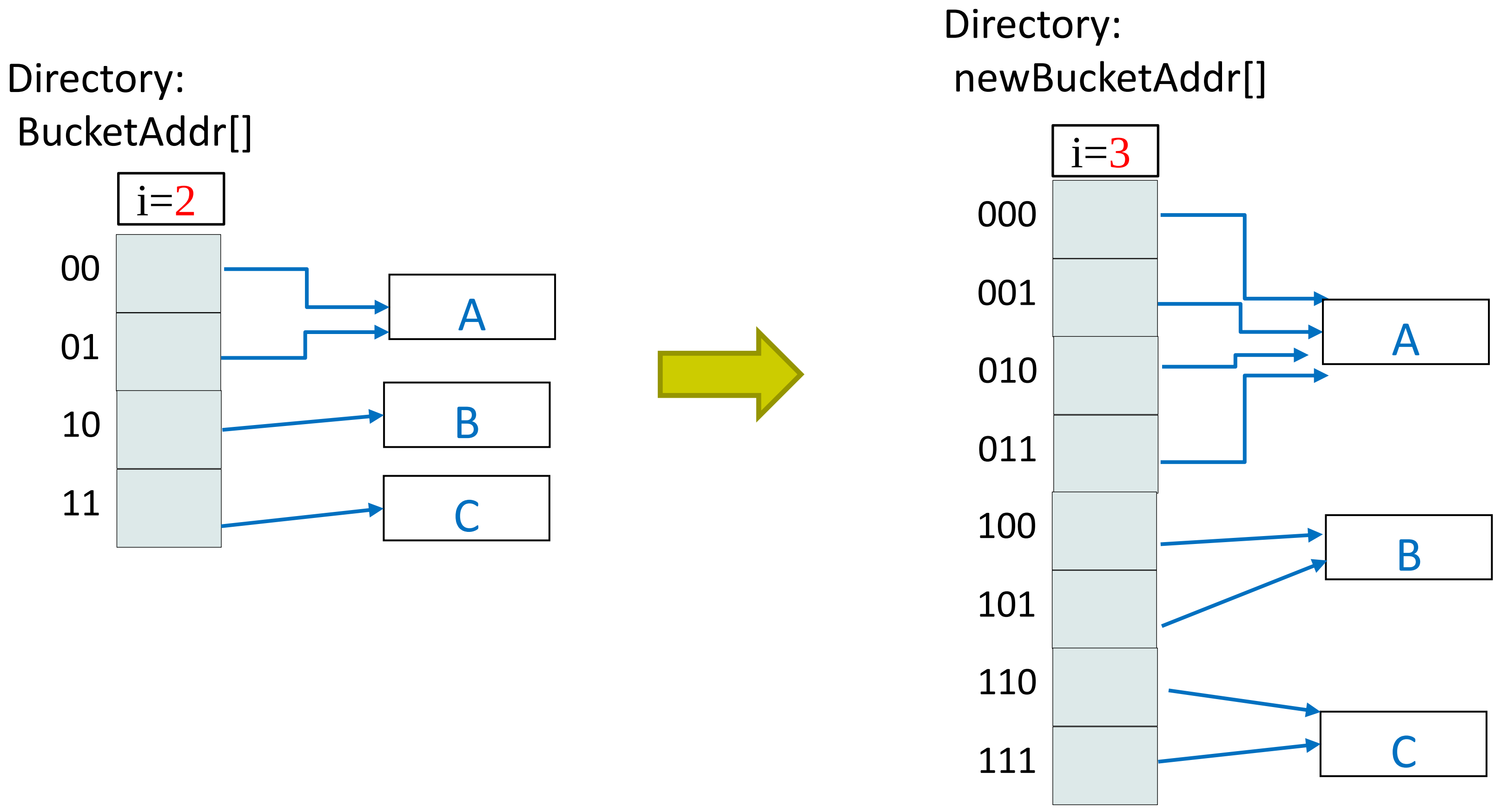
- Example2
  - Assume directory used 5 bit address
    - The range for the new bucket would start with 01000 and end with 01111
      - Covers all 5-bit addresses that share 01 as the first 2 bits
- Main logic of NewRange()
  - Starts by finding shared address bits
  - Start Range
    - Fills the address out with 0s until we have the number of bits used in the directory
  - End Rage
    - Fills the address out with 1

Bucket::NewRange (3/3)

```
// make a range for the new split bucket
int Bucket::NewRange (int & newStart, int & newEnd)
{
    int sharedAddr = MakeAddress(Keys[0], Depth);
    int bitsToFill = Dir.Depth - (Depth + 1);
    newStart = (sharedAddr << 1) | 1;
    newEnd = newStart;
    for (int j = 0; j < bitsToFill; j++)
    {
        newStart = newStart << 1;
        newEnd = (newEnd << 1) | 1;
    }
    return 1;
}
```

Directory::DoubleSize() (1/2)

```
newBucketAddr[2*i] = BucketAddr[i];
newBucketAddr[2*i+1] = BucketAddr[i];
```



Directory::DoubleSize() (2/2)

- Implementation

```
// double the size of the directory
int Directory::DoubleSize ()
{
    int newSize = 2 * NumCells;
    int * newBucketAddr = new int[newSize];
    for (int i = 0; i < NumCells; i++)
    {
        newBucketAddr[2*i] = BucketAddr[i];
        newBucketAddr[2*i+1] = BucketAddr[i];
    }
    delete BucketAddr;
    BucketAddr = newBucketAddr;
    Depth ++;
    NumCells = newSize;
    return 1;
}
```

Calculates the new directory size

Allocates the required memory

Writes the information from each old directory cell into two successive cells in the new directory

Freeing old space associated with BufferAddrs

Renaming the new space as the BufferAddrs

Directory::InsertBucket

- Attach a bucket address across a range of directory cell
- Invoked by Bucket::Split()
  - After NewRange()
  - InsertBucket(newBuckter->Addr, newStart, newEnd)

```
// change cells to refer to this bucket
int Directory::InsertBucket (int bucketAddr, int first, int last)
{
    for (int i = first; i <= last; i++)
        BucketAddr[i] = bucketAddr;
    return 1;
}
```

Bucket::Redistribute()

- Invoked by Bucket::Split()

```
int Bucket::Redistribute (Bucket & newBucket)
{
    // check each key in this bucket
    for (int i = NumKeys - 1; i >= 0; i--)
    {
        int bucketAddr = Dir.Find (Keys[i]); // look up the bucket
        if (bucketAddr != BucketAddr)// key belongs in the new bucket
        {
            newBucket.TextIndex::Insert (Keys[i], RecAddrs[i]);
            TextIndex::Remove (Keys[i]); // delete key from this bucket
        }
    }
    return 1;
}
```

Q&A