Storage and Indexing

Harvard Extension School

Cody Doucette, Ph.D.

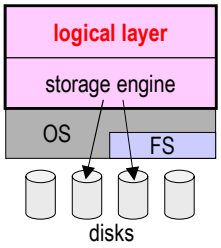
Lecture designed by David G. Sullivan

Accessing the Disk

- Data is arranged on disk in units called *blocks*.
  - typically fairly large (e.g., 4K or 8K)
- Relatively speaking, disk I/O is very expensive.
  - in the time it takes to read a single disk block, the processor could be executing millions of instructions!
- The DBMS tries to minimize the number of disk accesses.

Review: DBMS Architecture

- A DBMS can be viewed as a composition of two layers.
- At the bottom is the *storage layer* or *storage engine*, which takes care of storing and retrieving the data.
- Above that is the *logical layer*, which provides an abstract representation of the data.



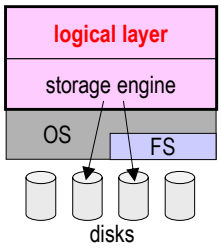
Logical-to-Physical Mapping

- The logical layer implements a mapping between:

  the *logical schema* of a database

  ↓ ↑

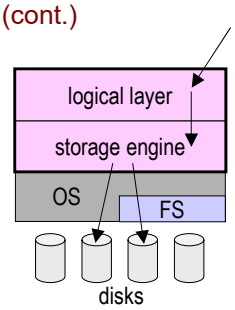
  its *physical representation*



- In the relational model, the schema includes:
  - attributes/columns, including their types
  - tuples/rows
  - relations/tables
- To be model-neutral, we'll use these terms instead:
  - *field* for an individual data value
  - *record* for a group of fields
  - *collection* for a group of records

Logical-to-Physical Mapping (cont.)

- A DBMS may use the filesystem, or it may bypass it and use its own disk manager.



- In either case, a DBMS may use units called *pages* that have a different size than the block size.
  - can be helpful in performance tuning

Logical-to-Physical Mapping (cont.)

- We'll consider:
  - how to map logical records to their physical representation
  - how to organize the records in a given collection
    - including the use of index structures
- Different approaches require different amounts of *metadata* – data about the data.
  - example: the types and lengths of the fields
  - *per-record* metadata – stored within each record
  - *per-collection* metadata – stored once for the entire collection
- Assumptions about data in the rest of this set of slides:
  - each character is stored using 1 byte
  - integer data values are stored using 4 bytes
  - integer metadata (e.g., offsets) are stored using 2 bytes

Fixed- or Variable-Length Records?

- This choice depends on:
  - the types of fields that the records contain
  - the number of fields per record, and whether it can vary
- Simple case: use fixed-length records when
  - all fields are fixed-length (e.g., CHAR or INTEGER),
  - there is a fixed number of fields per record

Fixed- or Variable-Length Records? (cont.)

- The choice is less straightforward when you have either:
  - variable-length fields (e.g., VARCHAR)
  - a variable number of fields per record (e.g., in XML)

Two options:

1. fixed-length records: always allocate the maximum possible length

...	comp sci	...
...	math	...

   - plusses and minuses:
     + less metadata is needed, because:
       - every record has the same length
       - a given field is in a consistent position within all records
     + changing a field's value doesn't change the record's length
       - thus, changes never necessitate moving the record
     – we waste space when a record has fields shorter than their max length, or is missing fields

Fixed- or Variable-Length Records? (cont.)

2. variable-length records: only allocate the space that each record actually needs

...	comp sci	...
...	math	...

   - plusses and minuses:
     - more metadata is needed in order to:
       - determine the boundaries between records
       - determine the locations of the fields in a given record
     - changing a field's value can change the record's length
       - thus, we may need to move the record
     + we *don't* waste space when a record has fields shorter than their max length, or is missing fields

Format of Fixed-Length Records

- With fixed-length records, we store the fields one after the other.
- If a fixed-length record contains a variable-length field:
  - allocate the max. length of the field
  - use a delimiter (**#** below) if the value is shorter than the max.
- Example:

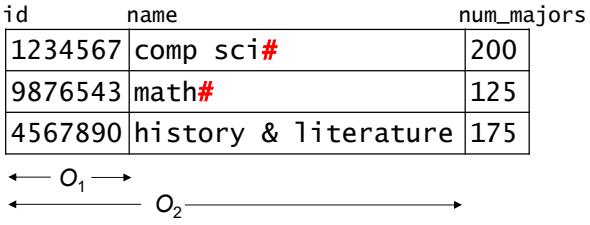
  `Dept(id CHAR(7), name VARCHAR(20), num_majors INT)`

id	name	num_majors
1234567	comp sci#	200
9876543	math#	125
4567890	history & literature	175

  - why doesn't `'history & literature'` need a delimiter?

Format of Fixed-Length Records (cont.)

- To find the position of a field, use *per-collection* metadata.
  - typically store the *offset* of each field (O_1 and O_2 below) – how many bytes the field is from the start of the record



- Notes:
  - the delimiters are the only *per-record* metadata
  - the records are indeed fixed-length – 31 bytes each!
    - 7 bytes for `id`, which is a `CHAR(7)`
    - 20 bytes for `name`, which is a `VARCHAR(20)`
    - 4 bytes for `num_majors`, which is an `INT`

Format of Variable-Length Records

- With variable-length records, we need *per-record* metadata to determine the locations of the fields.
- For simplicity, we'll assume all records in a given collection have the same # of fields.
- We'll look at how the following record would be stored:

```
     CHAR(7)       VARCHAR(20)     INT
('1234567', 'comp sci', 200)
```

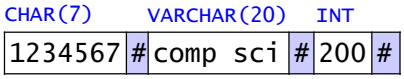
- We'll consider two types of operations:
  1. finding/extracting the value of a single field

```
SELECT num_majors
FROM Dept
WHERE name = 'comp sci';
```

  2. updating the value of a single field
     - its length may become smaller or larger

Format of Variable-Length Records (cont.)

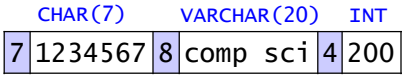
- Option 1: Terminate field values with a special delimiter character.



1. finding/extracting the value of a single field
   this is very inefficient; need to scan byte-by-byte to:
   - find the start of the field we're looking for
   - determine the length of its value (if it is variable-length)
2. updating the value of a single field
   if it changes in size, we need to shift the values after it,
   but we don't need to change their metadata

Format of Variable-Length Records (cont.)

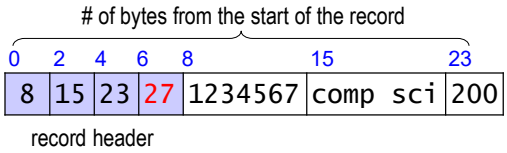
- Option 2: Precede each field by its length.



1. finding/extracting the value of a single field
   this is more efficient
   - can jump over fields, rather than scanning byte-by-byte
     (but may need to perform multiple jumps)
   - never need to scan to determine the length of a value
2. updating the value of a single field
   same as option 1

Format of Variable-Length Records (cont.)

- Option 3: Put offsets and other metadata in a record header.

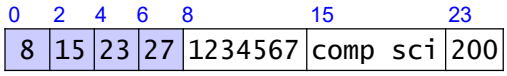


computing the offsets

- 3 fields in record → 4 offsets, each of which is a 2-byte int
- thus, the offsets take up 4*2 = 8 bytes
- $offset_0$ = 8, because $field_0$ comes right after the header
- $offset_1$ = 8 + len('1234567') = 8 + 7 = 15
- $offset_2$ = 15 + len('comp sci') = 15 + 8 = 23
- $offset_3$ = *offset of the end of the record*
  = 23 + 4 (since 200 an int) = 27
  We store this offset because it may be needed to compute the length of a field's value!

Format of Variable-Length Records (cont.)

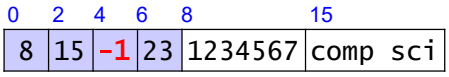
- Option 3 (cont.)



1. finding/extracting the value of a single field
   this representation is the most efficient. it allows us to:
   - jump directly to the field we're interested in
   - compute its length without scanning through its value
2. updating the value of a single field
   less efficient than options 1 and 2 if the length changes. why?

Representing Null Values

- Option 1: add an "out-of-band" value for every data type
  - con: need to increase the size of most data types, or reduce the range of possible values
- Option 2: use per-record metadata
  - example: use a special offset (e.g., -1)



Index Structures

- An index structure stores (key, value) pairs.
  - also known as a *dictionary* or *map*
  - we will sometimes refer to the (key, value) pairs as *items*
- The index allows us to more efficiently access a given record.
  - quickly find it based on a particular field
  - instead of scanning through the entire collection to find it
- A given collection of records may have multiple index structures:
  - one *clustered* or *primary* index
  - some number of *unclustered* or *secondary* indices

Clustered/Primary Index

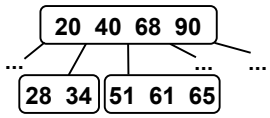
- The *clustered* index is the one that stores the full records.
  - also known as a *primary index,* because it is typically based on the primary key
- If the records are stored outside of an index structure, the resulting file is sometimes called a *heap file*.
  - managed somewhat like the heap memory region

Unclustered/Secondary Indices

- In addition to the clustered/primary index, there can be one or more *unclustered* indices based on other fields.
  - also known as *secondary indices*
- Example: *Customer(id, name, street, city, state, zip)*
  - primary index:
    (key, value) = (*id*, all of the remaining fields in the record)
  - a secondary index to enable quick searches by name
    (key, value) = (*name*, *id*) *does not include the other fields!*
- We need two lookups when we start with the secondary index.
  - example: looking for Ted Codd's zip code
  - search for `'Ted Codd'` in the secondary index
    ➔ `'123456'` (his id)
  - search for `'123456'` in the primary index
    ➔ his full record, including his zip code

B-Trees

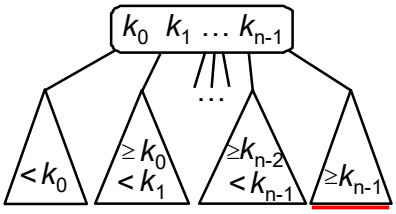
- A B-tree of order *m* is a tree in which each node has:
  - at most 2*m* items (and, for internal nodes, 2*m* + 1 children)
  - at least *m* items (and, for internal nodes, *m* + 1 children)
  - exception: the root node may have as few as 1 item
- Example: a B-tree of order 2



(we're just showing the keys)

- A B-tree has *perfect balance*: all paths from the root node to a leaf node have the same length.

Search in B-Trees

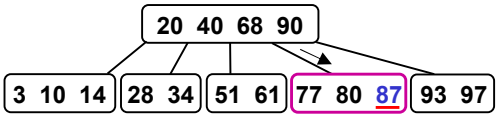


each triangle is a subtree

- A B-tree is a *search tree*.
  - like a binary search tree, but can have more keys per node
- When searching for an item whose key is *k*, we never need to enter more than one of the subtrees of a node.

Search in B-Trees (cont.)

- Example: search for the item whose key is 87



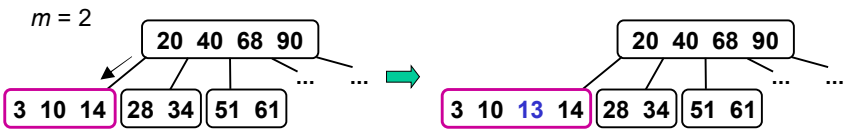
- Here's pseudocode for the algorithm:

```
search(key, node) {
   if (node == null) return null;
   i = 0;
   while (i < node.numkeys  &&  node.key[i] < key)
      i++;
   if (i == node.numkeys || node.key[i] != key)
      return search(key, node.child[i]);
   else       // node.key[i] == key
      return node.data[i];
}
```

Insertion in B-Trees

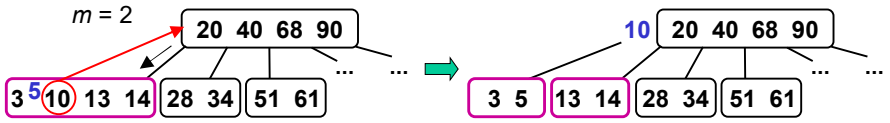
- Algorithm for inserting an item with a key *k*:
  - search for *k* until you reach a leaf node
  - if the leaf node has fewer than 2*m* items, add the new item to the leaf node
  - else *split the node*, dividing up the 2*m* + 1 items:
    - the first/smallest *m* items remain in the original node
    - the last/largest *m* items go in a new node
    - send the middle item up and insert it (and a pointer to the new node) in the parent

- Example of an insertion without a split: insert 13

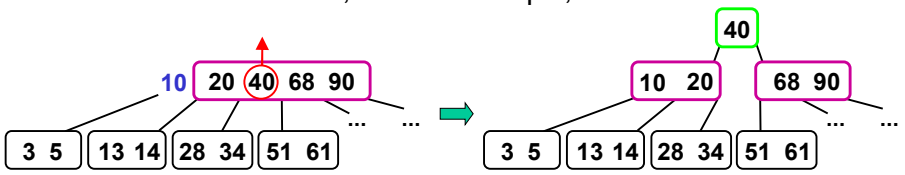


Splits in B-Trees

- Insert 5 into the result of the previous insertion:

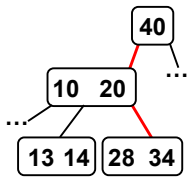


- The middle item (the 10) is sent up to the root.
  The root has no room, so it is also split, and a new root is formed:



- Splitting the root increases the tree's height by 1, but it remains balanced! This is only way the height increases.
- When an internal node is split, its $2m + 2$ pointers are split evenly between the original node and the new node.

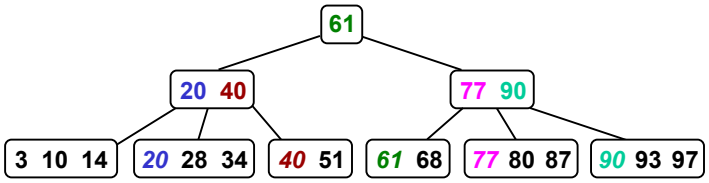
Other Details of B-Trees



- Each node in the tree corresponds to one page in the corresponding index file.
  - child pointers = page numbers
- Efficiency: In the worst case, searching for an item involves traversing a single path from the root to a leaf node.
  - # of nodes accessed <= tree height + 1
  - each internal node has at least m children
    - → tree height <= $\log_m n$, where n = # of items
    - → search and insertion are $O(\log_m n)$
- To minimize disk I/O, make *m* as large as possible.
  - but not too large!
  - if *m* is too large, can end up with items that don't fit on the page and are thus stored in separate *overflow pages*

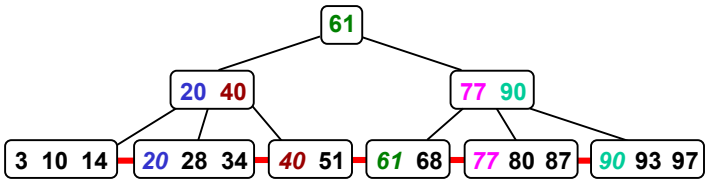
B+Trees

- A B+tree is a B-tree variant in which:
  - data items are only found in the leaf nodes
  - internal nodes contain only keys and child pointers
  - an item's key may appear in a leaf node *and* an internal node
- Example: a B+tree of order 2



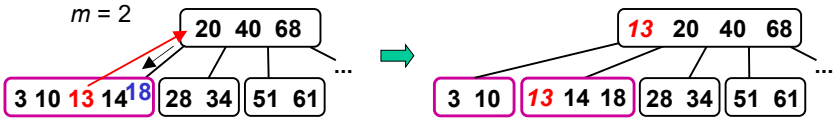
B+Trees (cont.)

- Advantages:
  - there's more room in the internal nodes for child pointers
    - why is this beneficial?
  - because all items are in leaf nodes, we can link the leaves together to improve the efficiency of operations that involve scanning the items in key order (e.g., range searches)



Differences in the Algorithms for B+Trees

- When searching, we keep going until we reach a leaf node, even if we see the key in an internal node.
- When splitting a leaf node with $2m + 1$ items:
  - the first m items remain in the original node as before
  - *all* of the remaining $m + 1$ items are put in the new node, *including the middle item*
  - the *key* of the middle item is *copied* into the parent
    - why can't we move up the entire item as before?
- Example: insert 18



Differences in the Algorithms for B+Trees (cont.)

- Splitting an internal node is the same as before, but with keys only:
  - first m keys stay in original node, last m keys go to new node
  - middle key is sent up to parent (not copied)

Deletion in B-Trees and B+Trees

- Search for the item and remove it.
- If a node N ends up with fewer than *m* items, do one of the following:
  - if a sibling node has more than *m* items, take items from it and add them to N
  - if the sibling node only has *m* items, *merge* N with the sibling
- If the key of the removed item is in an internal node, *don't* remove it from the internal node.
  - we need the key to navigate to the node's children
  - can remove when the associated child node is merged with a sibling
- Some systems don't worry about nodes with too few items.
  - assume items will be added again eventually

Ideal Case: Searching = Indexing

- The ideal index structure would be one in which:
  key of data item = the page number where the item is stored
- In most real-world problems, we can't do this.
  - the key values may not be integers
  - we can't afford to give each key value its own page
- To get something close to the ideal, we perform *hashing*:
  - use a *hash function* to convert the keys to page numbers

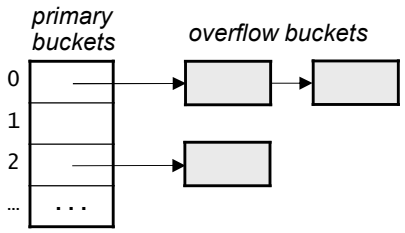
    `h('hello') → 5`
- The resulting index structure is known as a *hash table*.

Hash Tables: In-Memory vs. On-Disk

- In-memory:
  - the hash value is used as an index into an array
  - depending on the approach you're taking, a given array element may only hold one item
  - need to deal with *collisions* = two values hashed to same index
- On-disk:
  - the hash value tells you which *page* the item should be on
  - because pages are large, each page serves as a *bucket* that stores multiple items
  - need to deal with full buckets

Static vs. Dynamic Hashing

- In *static hashing*, the number of buckets never changes.
  - if a bucket becomes full, we use overflow buckets/pages

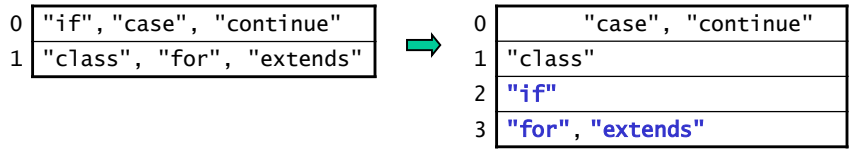


  - why is this problematic?
- In *dynamic hashing*, the number of buckets can grow over time.
  - can be expensive if you're not careful!

A Simplistic Approach to Dynamic Hashing

- Assume that:
  - we're using keys that are strings
  - h(key) = number of characters in key
  - we use mod (%) to ensure we get a valid bucket number:

    bucket index = h(key) % number of buckets
- When the hash table gets to be too full:
  - double the number of buckets
  - rehash **all** existing items. why?



Linear Hashing

- It does ***not*** use the modulus to determine the bucket index.
- Rather, it treats the hash value as a binary number, and it uses the i *rightmost* bits of that number:

  i = ceil($\log_2$n) where n is the current number of buckets
  - example: n = 3 → i = ceil($\log_2$3) = 2
- If there's a bucket with the index given by the i rightmost bits, put the key there.

```
    h("if") = 2 = 00000010
  h("case") = 4 = 00000100
 h("class") = ?
h("continue") = ?
```

00 = 0	"case"
01 = 1	
10 = 2	"if"

- If not, use the bucket specified by the rightmost i – 1 bits

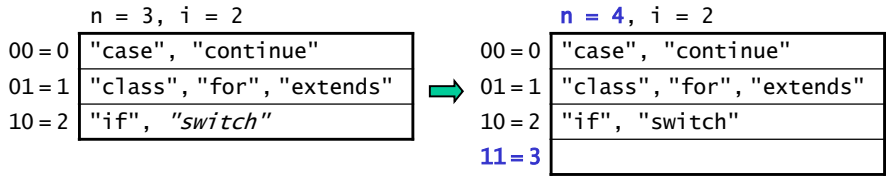
```
    h("for") = 3 = 00000011   (11 = 3 is too big, so use 1)
h("extends") = ?
```

Linear Hashing: Adding a Bucket

- In linear hashing, we keep track of three values:
  - n, the number of buckets
  - i, the number of bits used to assign keys to buckets
  - f, some measure of how full the buckets are
- When f exceeds some threshold, we:
  - add **only one** new bucket
  - increment n and update i as needed
  - rehash/move keys as needed
- We only need to rehash the keys in **one** of the old buckets!
  - if the new bucket's binary index is 1xyz (xyz = arbitrary bits), rehash the bucket with binary index 0xyz
- Linear hashing has to grow the table more often, but each new addition takes very little work.

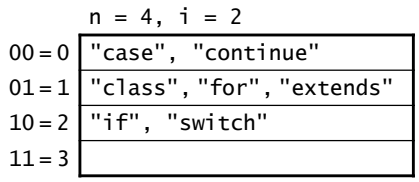
Example of Adding a Bucket

- Assume that:
  - our measure of fullness, f = # of items in hash table
  - we add a bucket when f > 2*n
- Continuing with our previous example:
  - n = 3; f = 6 = 2*3, so we're at the threshold
  - adding "switch" exceeds the threshold, so we:
    - add a new bucket whose index = 3 = 11 in binary
    - increment n to 4 → i = ceil($\log_2 4$) = 2 *(unchanged)*



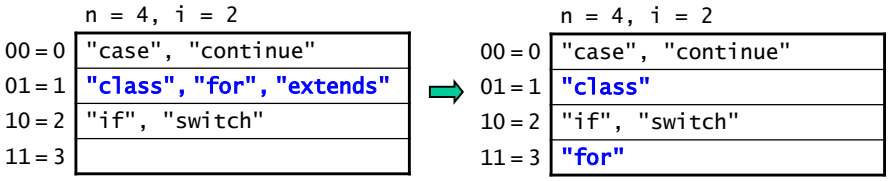
Example of Adding a Bucket (cont.)

- Which previous bucket do we need to rehash?
  - new bucket has a binary index of 11
  - because this bucket wasn't there before,
    items that should now be in 11 were originally put in 01
    (using the rightmost i – 1 bits)



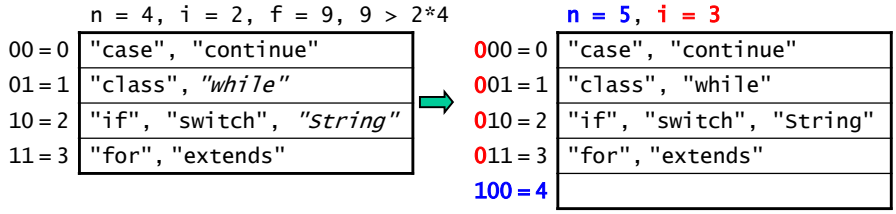
Example of Adding a Bucket (cont.)

- Which previous bucket do we need to rehash?
  - new bucket has a binary index of 11
  - because this bucket wasn't there before, items that should now be in 11 were originally put in 01 (using the rightmost i – 1 bits)
  - thus, we rehash bucket 01:
    - h("class") = 5 = 0000010**1** (leave where it is)
    - h("for") = 3 = 0000001**1** (move to new bucket)
    - h("extends") = 7 = 0000011**1**



Additional Details

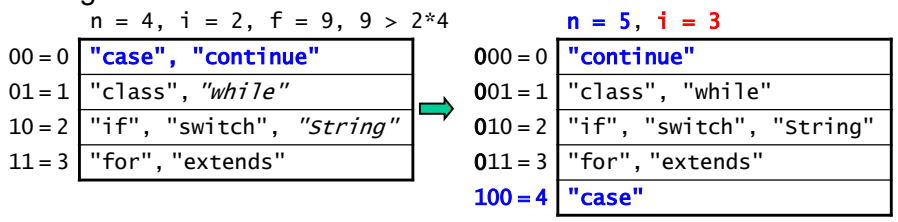
- If the number of buckets exceeds 2^i, we increment i and begin using one additional bit.



which bucket should be rehashed?

Additional Details

- If the number of buckets exceeds 2^i, we increment i and begin using one additional bit.



- The process of adding a bucket is sometimes referred to as *splitting* a bucket.
  - example: adding bucket 4 <==> splitting bucket 0
    because some of 0's items may get moved to bucket 4
- The split bucket:
  - may retain all, some, or none of its items
  - may not be as full as other buckets
    - thus, linear hashing still allows for overflow buckets as needed

More Examples

- Assume again that we add a bucket whenever the # of items exceeds 2n.
- What will the table below look like after inserting the following sequence of keys? (assume no overflow buckets are needed)

```
"toString":  h("toString")  = ?
```

n = 5, i = 3

000 = 0	"continue"
001 = 1	"class", "while"
010 = 2	"if", "switch", "String"
011 = 3	"for", "extends"
100 = 4	"case"

Hash Table Efficiency

- In the best case, search and insertion require at most one disk access.
- In the worst case, search and insertion require *k* accesses, where *k* is the length of the largest bucket chain.
- Dynamic hashing can keep the worst case from being too bad.

Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.
- The items are not ordered by key. As a result, we can't easily:
  - access the records in sorted order
  - perform a range search
  - perform a rank search – get the kth largest value of some field

  We *can* do all of these things with a B-tree / B+tree.

Which Index Structure Should You Choose?

- Recently accessed pages are stored in a *cache* in memory.
- Working set = collection of frequently accessed pages
- If the working set fits in the cache, use a B-tree / B+tree.
  - efficiently supports a wider range of queries (see last slide)
- If the working set can't fit in memory:
  - choose a B-tree/B+tree if the workload exhibits *locality*
    - locality = a query for a key is often followed by a query for a key that is nearby in the space of keys
    - because the items are sorted by key, the neighbor will be in the cache
  - choose a hash table if the working set is very large
    - uses less space for "bookkeeping" (pointers, etc.), and can thus fit more of the working set in the cache
    - fewer operations are needed before going to disk