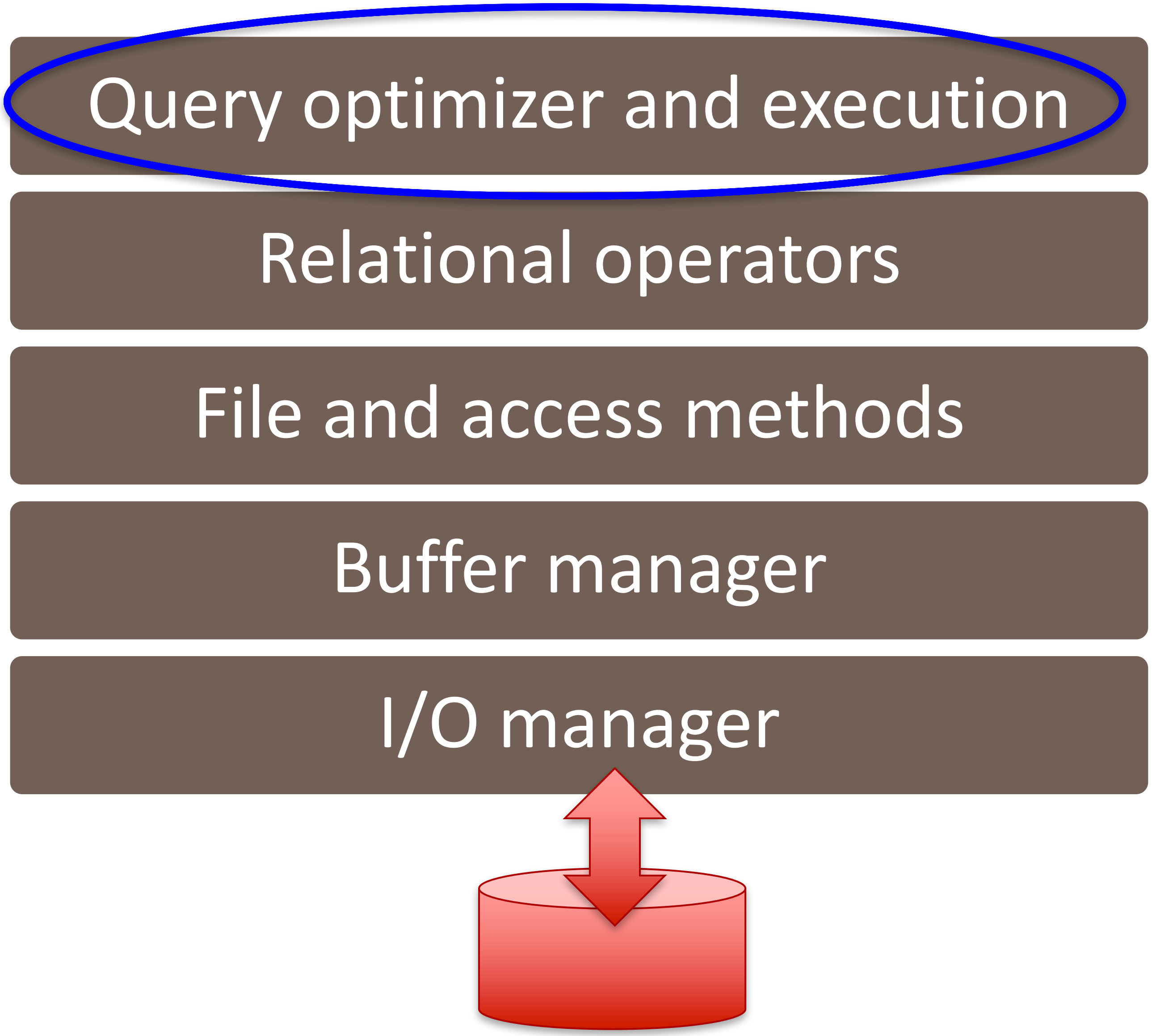
BadgerDB::Btree

- Goal: Build key components of a RDBMS
  - First hand experience building the internals of a simple database system
  - And have some fun doing so!
- Two parts
  - *Buffer manager [✓]*
  - **B+tree** *(Due Date : Mar 27 by 2PM)*
    - ***First class day after the Spring break***

All projects are individual assignments

Structure of Database



Query optimizer and execution

Relational operators

File and access methods

Buffer manager

I/O manager

Plan for today

- Review of C++ templates and helpful functions
  - `memset`, `memcpy` and `reinterpret_cast`
- B+ tree: insertion
- <break>
- BadgerDB::Btree
  - Project specifications
  - Code
- Q&A

C++ templates

Human misery of duplicate codes

- Suppose you write a function printData:

```
void printData(int value) {
      std::cout << "The value is "<< value;
}
```

- later you want to print double and std::string

```
void printData(double value) {
      std::cout << "The value is "<< value;
}

void printData(std::string value) {
      std::cout << "The value is "<< value;
}
```

And Stroustrup said - let there be templates

-
```
template<typename T>
void printData(T value) {
        std::cout << "The value is ” << value;
}
```

And Stroustrup said - let there be templates

-
```
template<typename T>
void printData(T value) {
        std::cout << "The value is ” << value;
}
```

Template semantics

- The syntax is simple:

  `template< typename name ⇔ class name >`

- Function templates
- Class templates

Function templates

-
  ```
  template<typename T>
  void func() {
  }

  int main() {
      func<int>();
      func<double>();
  }
  ```

Function templates

-
  ```
  template<typename T>
  void func() {
  }

  int main() {
      func<int>();
      func<double>();
  }
  ```

```
template<typename T>
void func(T value) {
}
template<typename T, typename U>
T func2(U value) {
    return T(value);
}

int main() {
    // T=int
    func(3);
    // T=double
     func(3.5);
    // T=int, U=double
    func2(3.5);
    // T=std::vector, U=int
    func2<std::vector> (5);
    // specify both T and U
    // T=std::vector, U=int
    func2<std::vector, int>(5.7);
}
```

Class templates

- Also works on `structs`

```
template<typename T, int i>
struct FixedArray {
    T data[i];
};

FixedArray<int, 3> a;
// array of 3 integers
```

Class templates

- Also works on `structs`

```
template<typename T, int i>
struct FixedArray {
    T data[i];
};

FixedArray<int, 3> a;
// array of 3 integers
```

```
template<typename T>
class MyClass {
};

template<typename T1, typename T2=int>
class MyClass{};

// specify all parameters
MyClass<double, std::string> mc1;

// default value for T2
MyClass<int> mc4;
```

Template requirements

- Templates implicitly impose requirements on their parameters
- Type T has to be:
  - *Copy-Constructible* if

    `T a(b);`
  - *Assignable i.e.* defines `operator=()` if:

    `a = b;`
  - etc
- For this project: operations such as `a < b` could mean different for `int` and `std::string`

Pointers and arrays

Pointers and arrays

- arrays work very much like pointers to their first elements

```
int myarray [20];
int * mypointer;
```

Pointers and arrays

- arrays work very much like pointers to their first elements

```
int myarray [20];
int * mypointer;
```

Can you do?

```
mypointer = myarray;
```

Pointers and arrays

- arrays work very much like pointers to their first elements

```
int myarray [20];
int * mypointer;
```

Can you do?

```
mypointer = myarray;
```

```
Can you do?
myarray = mypointer;
```

Pointers and arrays

- arrays work very much like pointers to their first elements

```
int myarray [20];
int * mypointer;
```

Can you do?

```
mypointer = myarray;      ← Yes
```

```
Can you do?
myarray = mypointer;      ← No
```

Example

```
// more pointers
#include <iostream>
 using namespace std;
 int main () {
    int numbers[5];
    int * p;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << ", ";
    return 0;
}
```

Example

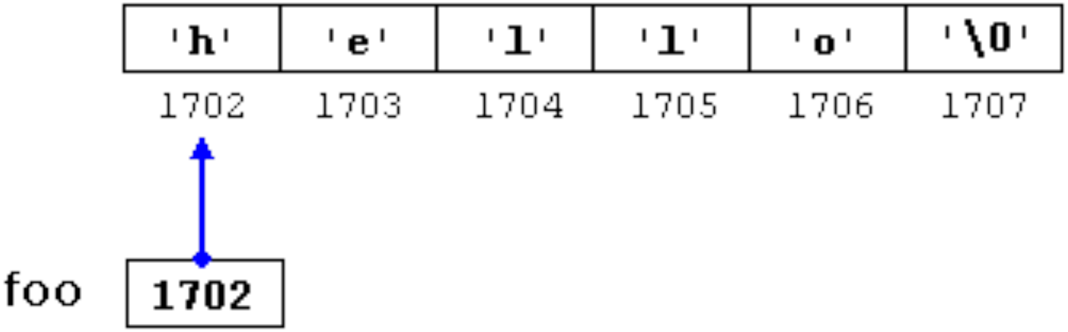
```
// more pointers
#include <iostream>
 using namespace std;
 int main () {
    int numbers[5];
    int * p;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << ", ";
    return 0;
}
```

Prints:

```
10, 20, 30, 40, 50,
```

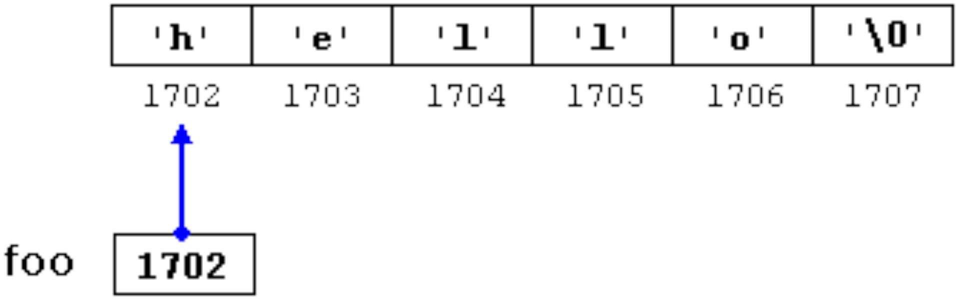
Pointers and string literal

- const char * foo = "hello";



Pointers and string literal

- const char * foo = "hello";



- Foo contains the value 1702, and not 'h', nor "hello“
- What is the output of?

  *(foo+4)

  foo[4]

C/C++ helpful fuctions

memset

- `void * memset ( void * ptr, int value, size_t num );`
- Fill block of memory
  - Sets the first *num* bytes of the block of memory pointed by *ptr* to the specified *value* (interpreted as an unsigned char)

```
int main () {
    char str[] = "almost every programmer should know memset!";
     memset (str,'-',6);
    print(str);
    return 0;
 }
```

memcpy

- void * memcpy (void * dest, const void * source, size_t num);
- Copy block of memory
  - Copies the values of *num* bytes from the location pointed to by *source* directly to the memory block pointed to by *destination*.
- std::memcpy is meant to be the fastest library routine for memory-to-memory copy (usually more efficient than std::strcpy)

```
struct { char name[40]; int age; } person, person_copy;

int main () {
    char myname[] = ”Bucky Badger";
    /* using memcpy to copy string: */
    memcpy ( person.name, myname, strlen(myname)+1 );
    /* using memcpy to copy structure: */
    memcpy ( &person_copy, &person, sizeof(person) );
    return 0;
}
```

reinterpret_cast<new_type>(expr)

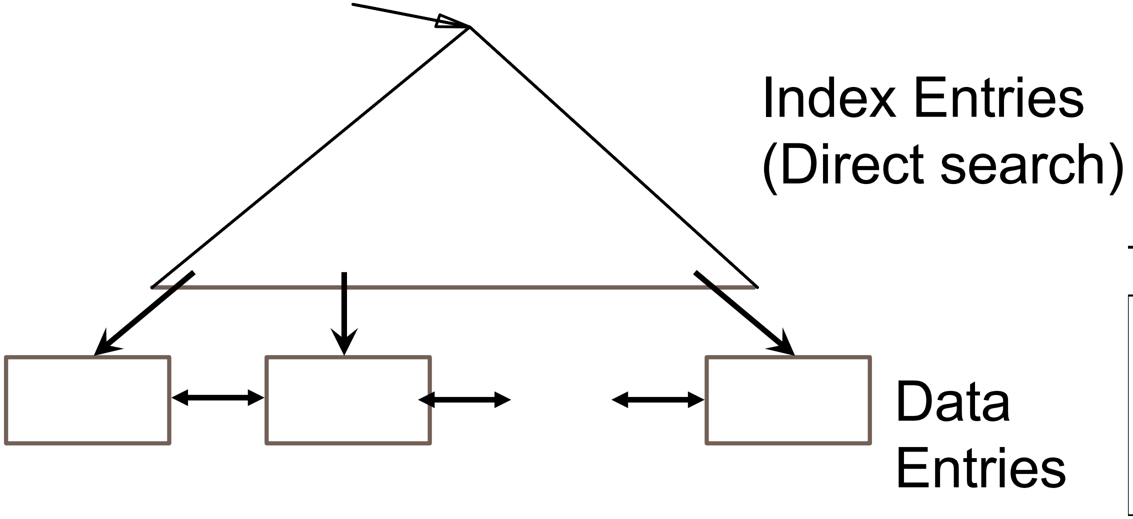
- reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes.
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

```
int main () {
    int  i  =  7;
    int* p1 = reinterpret_cast<int*>(&i);
    assert(p1 ==  &i);
    // type aliasing through pointer
    char* p2 = reinterpret_cast<char*>(&i);
    // type aliasing through reference
    reinterpret_cast<unsigned int&>(i) = 42;
    std::cout << i << '\n';
}
```

B+ tree

B+ tree

- Occupancy (d)
  - Minimum 50% occupancy (except for root)
  - Each node contains d <= m <= 2d entries.



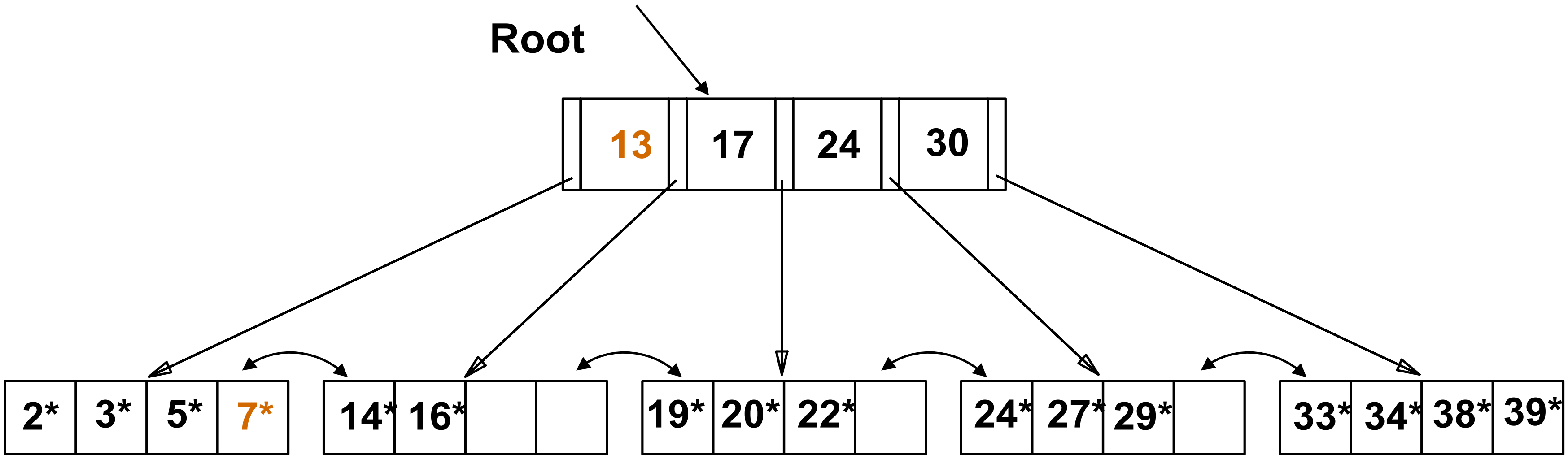
Index Entries
Entries in the index
(i.e. non-leaf) pages:
(search key value, pageid)

Data Entries
Entries in the leaf pages:
(search key value, recordid)

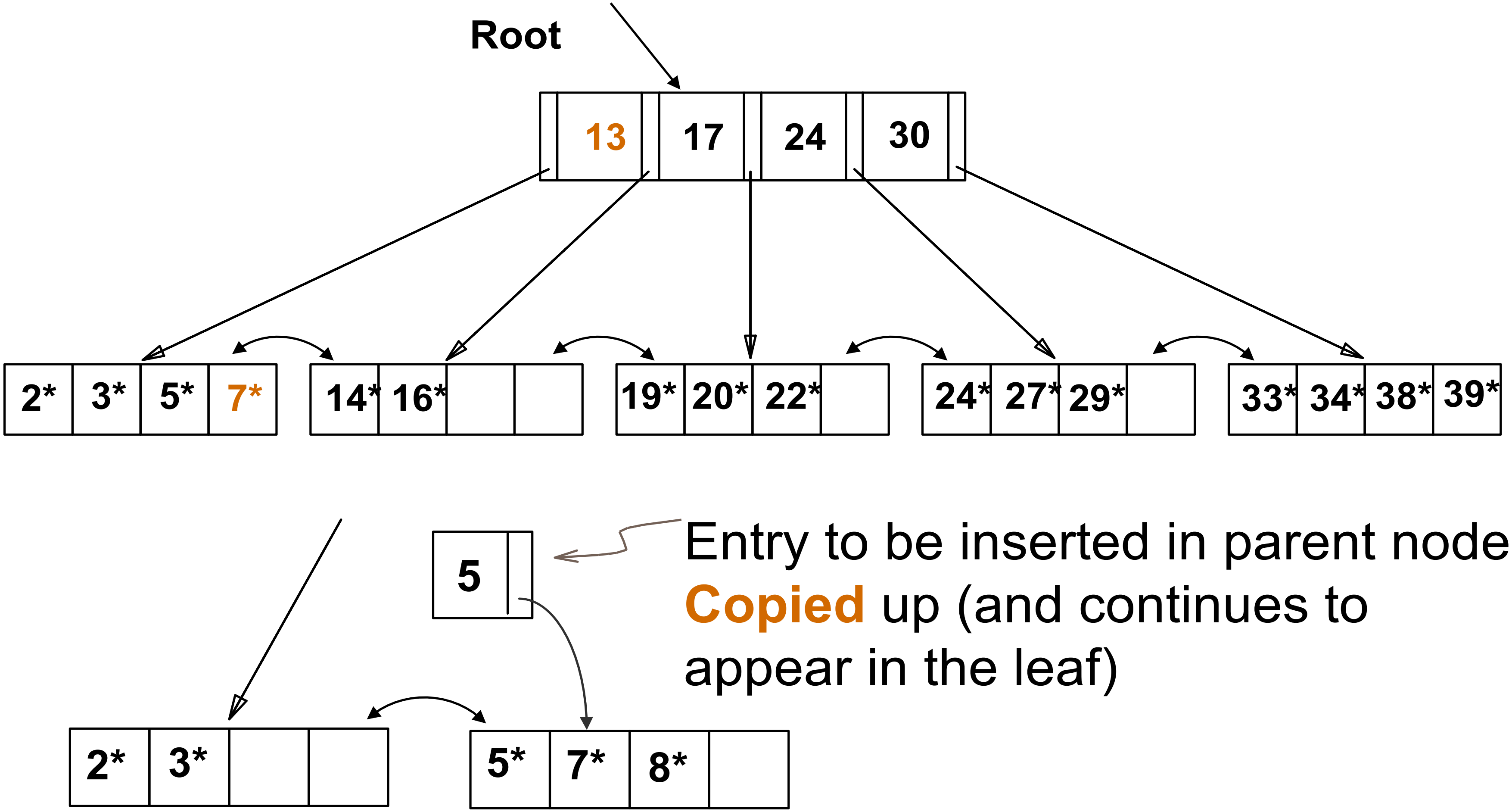
B+-Tree: Inserting a Data Entry

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must ***split*** *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split non-leaf node, redistribute entries evenly, but **pushing up** the middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Inserting 8* into B+ Tree



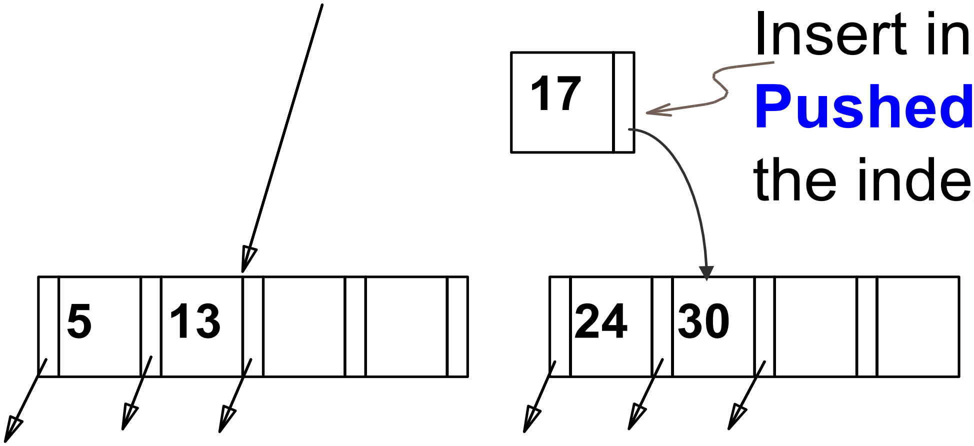
Inserting 8* into B+ Tree



Entry to be inserted in parent node

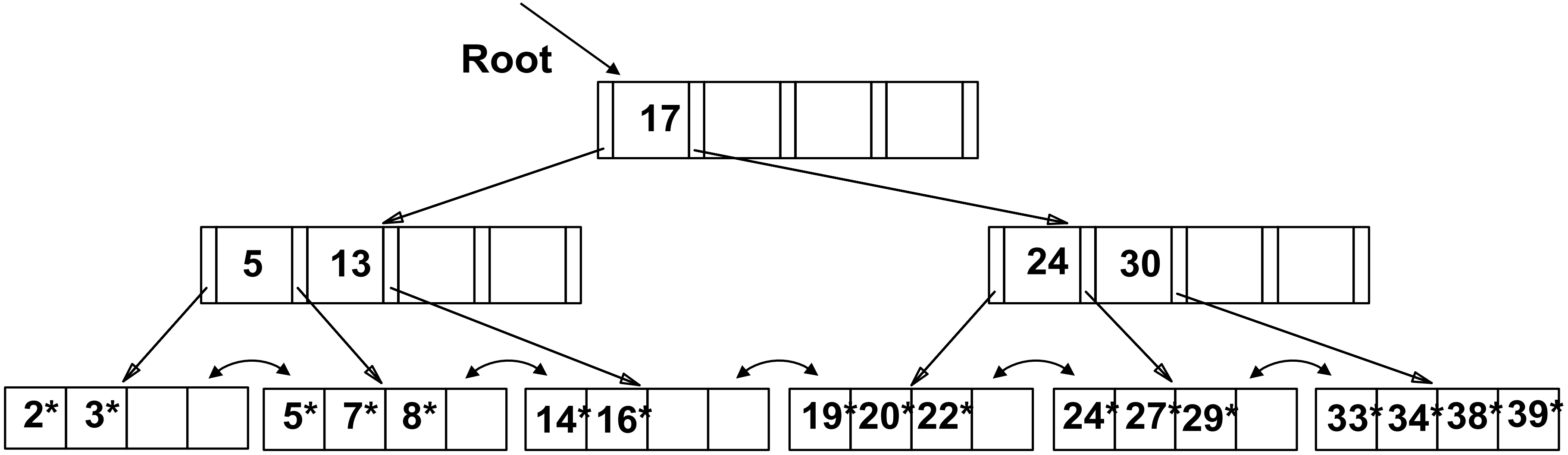
Copied up (and continues to appear in the leaf)

Inserting 8* into B+ Tree



Insert in parent node.
Pushed up (and only appears once in the index)

Inserting 8* into B+ Tree



- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  - Sibling: immediately to left or right, and same parent

5 mins break

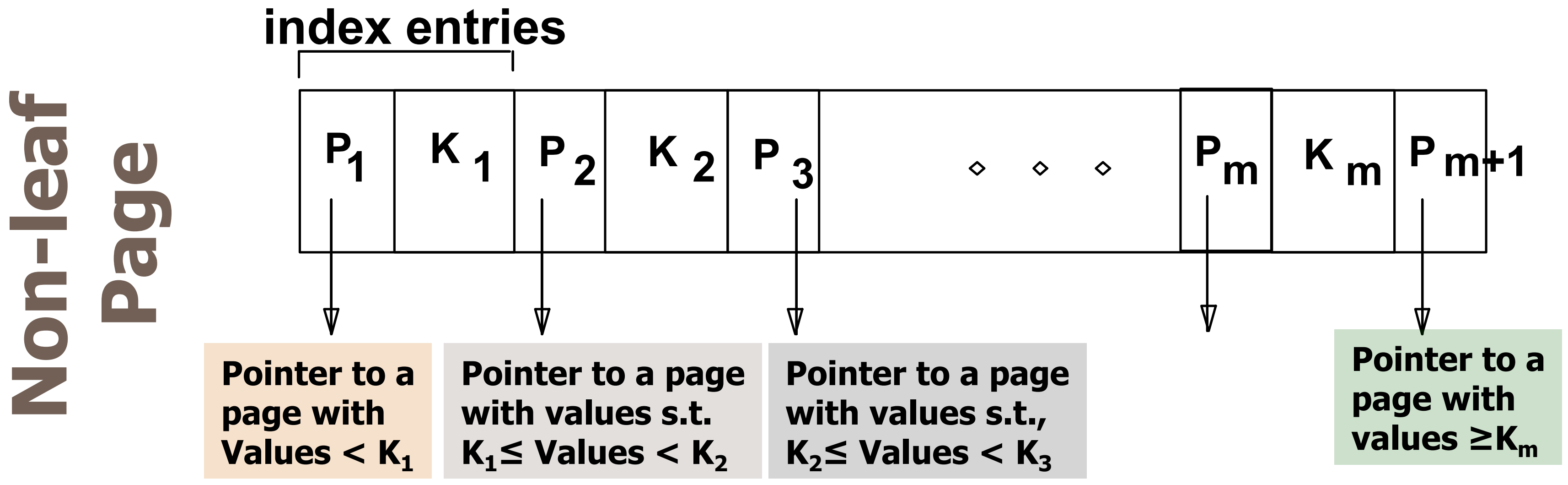
https://www.youtube.com/watch?v=AxSdWhkMB_A

BadgerDB: Btree

After you untar the project:

- btree.h: Add your own methods and structures as you see fit but don't modify the public methods that we have specified.
- btree.cpp: Implement the methods we specified and any others you choose to add.
- file.h(cpp): Implements the PageFile and BlobFile classes.
- main.cpp: Use to test your implementation. Add your own tests here or in a separate file. This file has code to show how to use the FileScan and BTreeIdnex classes.
- page.h(cpp): Implements the Page class.
- buffer.h(cpp), bufHashTbl.h(cpp): Implementation of the buffer manager.
- Exceptions/* : Implementation of exception classes that you might need.
- Makefile – makefile for this project.

B+-tree Page Format

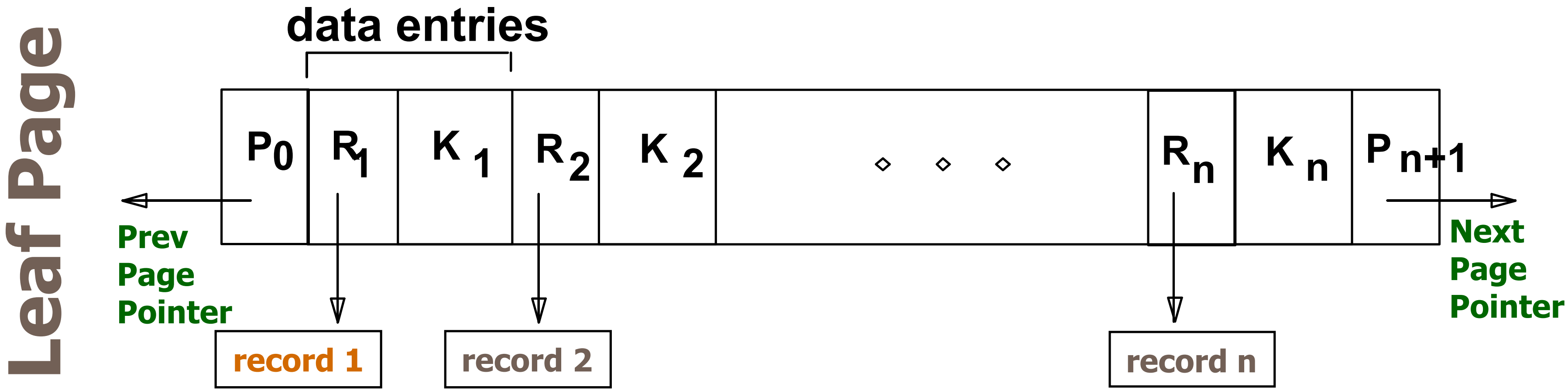


Pointer to a page with Values < K_1

Pointer to a page with values s.t. $K_1 \leq$ Values < K_2

Pointer to a page with values s.t., $K_2 \leq$ Values < K_3

Pointer to a page with values $\geq K_m$

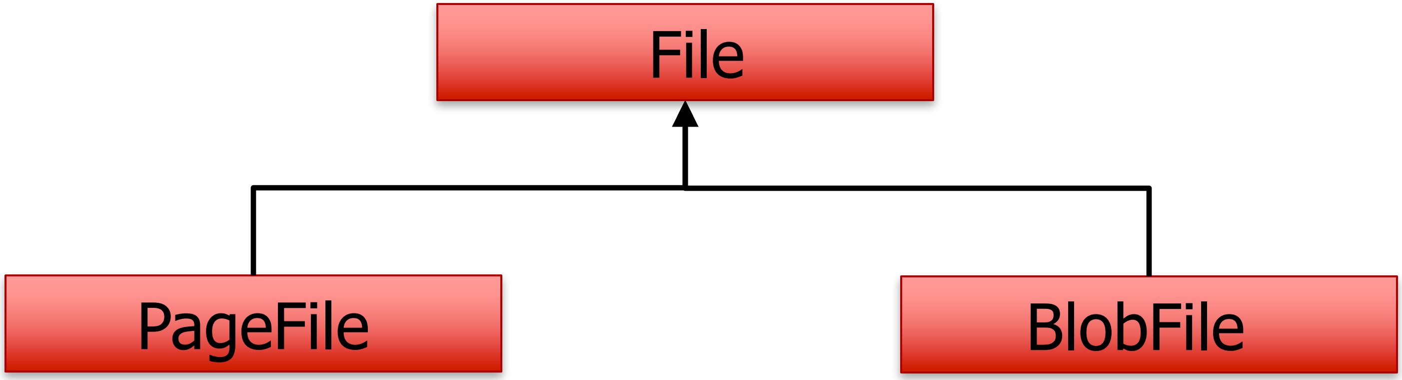


Prev Page Pointer

Next Page Pointer

Index

- the *index* will store *data entries* in the form **`<key, rid>`** pair
- stored in a file that is separate from the data file
- i.e. the index file "points to" the data file where the actual records are stored



File

PageFile

- stores all the relations (actual data) as we did in the buffer manager assignment
- You don't actually use this one for this project

BlobFile

- pages in the file are not linked by *prevPage/nextPage* links
- treats the pages as blobs of 8KB size i.e does not require these pages to be valid objects of the Page class
- use the BlobFile to store the B+ index file
- every page in the file is a node from the B+tree
- we can modify these pages to suit the particular needs of the B+ tree index

FileScan class

- The `FileScan` class is used to scan records in a file.
- FileScan(const std::string &relationName, BufMgr *bufMgr)
  - The constructor takes the relationName and buffer manager instance
- `~FileScan()`
  - Shutsdown the scan and unpins any pinned pages.
- `void scanNext(RecordId& outRid)`
  - Returns (via the outRid parameter) the RecordId of the next record from the relation being scanned. It throws EndOfFileException() when the end of relation is reached.
- `std::string getRecord()`
  - Returns a pointer to the "current" record. The record is the one in a preceding scanNext() call.
- `void markDirty()`
  - You don't need this for this assignment

BadgerDB: B+Tree Index

Simplifications:

- assume that all records in a file have the same length (so for a given attribute its offset in the record is always the same).
- only needs to support single-attribute indexing
- the indexed attribute may be one of three data types: integer, double, or string
- in the case of a string, you can use the first 10 characters as the key in the B+-tree
- we will never insert two data entries into the index with the same key value

B+Tree Index: Constructor

If the index file already exists, open the file.
Else, create a new index file
Parameters:

const string & relationName	The name of the relation on which to build the index. The constructor should scan this relation (using FileScan) and insert entries for all the tuples in this relation into the index
String & outIndexName	The name of the index file; determine this name in the constructor as shown above, and return the name.
BufMgr *bufMgrIn	The instance of the global buffer manager.
const int attrByteOffset	The byte offset of the attribute in the tuple on which to build the index. For instance, if we are storing the following structure as a record in the original relation: And, we are building the index over the double d, then the attrByteOffset value is 0+offsetof(RECORD, i), where offsetof is the offset position provided by the standard C++ library "offsetoff".
const Datatype attrType	The data type of the attribute we are indexing. Note that the Datatype enumeration {INTEGER, DOUBLE, STRING} is defined in btree.h

B+Tree Index: insertEntry

insertEntry

inserts a new entry into the index using the pair <key, rid>.

Input to this function:

const void * key	A pointer to the value (integer/double/string) we want to insert.
const RecordId & rid	The corresponding record id of the tuple in the base relation.

B+Tree Index: insertEntry

insertEntry

inserts a new entry into the index using the pair <key, rid>.

Input to this function:

const void * key	A pointer to the value (integer/double/string) we want to insert.
const RecordId & rid	The corresponding record id of the tuple in the base relation.

You will be spending bulk of your time /code in this method

B+Tree Index: startScan

startScan

This method is used to begin a “filtered scan” of the index.
For e.g. if the method is called using arguments (“a”,GT,”d”,LTE), the scan should seek all entries greater than “a” and less than or equal to “d”.

Input to this function:

const void * lowValue	The low value to be tested.
const Operator lowOp	The operation to be used in testing the low range. You should only support GT and GTE here; anything else should throw BadOpcodesException.
const void * highValue	The high value to be tested.
const Operator highOp	The operation to be used in testing the high range. You should only support LT and LTE here; anything else should throw BadOpcodesException.

B+Tree Index: scanNext

- **scanNext**
- fetches the record id of the next tuple that matches the scan criteria. If the scan has reached the end, then it should throw the exception IndexScanCompletedException

RecordId & outRid	output value this is the record id of the next entry that matches the scan filter set in startScan.

B+Tree Index: endScan

- **endScan**
- terminates the current scan and *unpins* all the pages that have been pinned for the purpose of the scan
- throws ScanNotInitializedException if called before a successful startScan call.

Implementation notes

- call the buffer manager to read/write pages
- don't keep the pages pinned in the buffer pool unless you need to
- For the scan methods, you will need to remember the "state" of the scan specified during the startScan
- insert does **not** need to redistribute entries
- At the leaf level, you do not need to store pointers to both siblings. The leaf nodes only point to the "next" (the right) sibling

FAQs

- **How do I get started?**

```
if the index file does not exist:
    create new BlobFile.
    allocate new meta page
    allocate new root page
    populate 'IndexMetaInfo' with the rootpage num
    scan records and insert into the BTree
else
    read the first page from the file - which is the meta node
    get the root page num from the meta node
    read the root page (bufManager->readPage(file, rootpageNum, out_root_page)
    once you have the root node, you can traverse down the tree
```

FAQs

- how to check whether an index file exists?
  - See file.h:

`static bool exists(const std::string& filename)`

FAQs

- How do I write a node to disk as a Page? e.g. how to write *IndexMetaInfo* node to the file?

⇒ You first need to allocate a Page using the bufferManager.

```
Page* metaPage;
bufManager->allocatePage(..., metaPage);
```

Then you can either cast it as IndexMetaInfo* and update it's parameter.
Another way is to first create and populate MetaIndexInfo node.
Then you can allocate a new Page* using bufferManager as above.
Then use 'memcpy' to copy tothe new Page:

```
memcpy(metaPage, &metaInfo,sizeof(IndexMetaInfo));
```

But you do not need to write it back as page **to disk explicitly. Buffer Manager does it for you.**
Remember project 2?

FAQs

- how to convert *Page* - that you read from the file to *Node* ?
  - you can cast e.g. using *reinterpret_cast*

Suggestions

- Start early
  - 1000+ lines of codes
- Try to finish before the spring break
  - No TA hours during the break
- Make incremental progress
  - Test aggressively