# CSE 250
# Lecture 39

## Final Review

Day 3

# Exam Details

- Where: NSC 225
- When: **7:15 PM**, Monday Dec 12
- Notes: 1 double-sided 8.5x11 “cheat sheet”
  - I strongly encourage you to use less

# Hash Tables

# Variations

- **Hash Table with Chaining**
  - ... but re-use empty hash buckets instead of chaining
    - **Hash Table with Open Addressing**
    - **Cuckoo Hashing** (Double Hashing)
  - ... but avoid bursty rehashing costs
    - **Dynamic Hashing**
  - ... but avoid O(N) iteration cost
    - **Linked Hash Table**

# Open Addressing

- insert(X)
  - While bucket hash(X)+i %N is occupied, i = i + 1
  - Insert at bucket hash(X)+i %N
- apply(X)
  - While bucket hash(X)+i %N is occupied
    - If the element at bucket hash(X)+i %N is X, return it
    - Otherwise i = i + 1
  - Element not found

# Open Addressing

- **Linear Probing**: Offset to hash(X)+ci for some constant c
- **Quadratic Probing**: Offset to hash(X)+$ci^2$ for some constant c
- Follow Probing Strategy to find the next bucket

- Runtime Costs
  - Chaining: Dominated by following chain
  - Open Addressing: Dominated by probing
- With a low enough $\alpha_{max}$, operations still O(1)

# Cuckoo Hashing

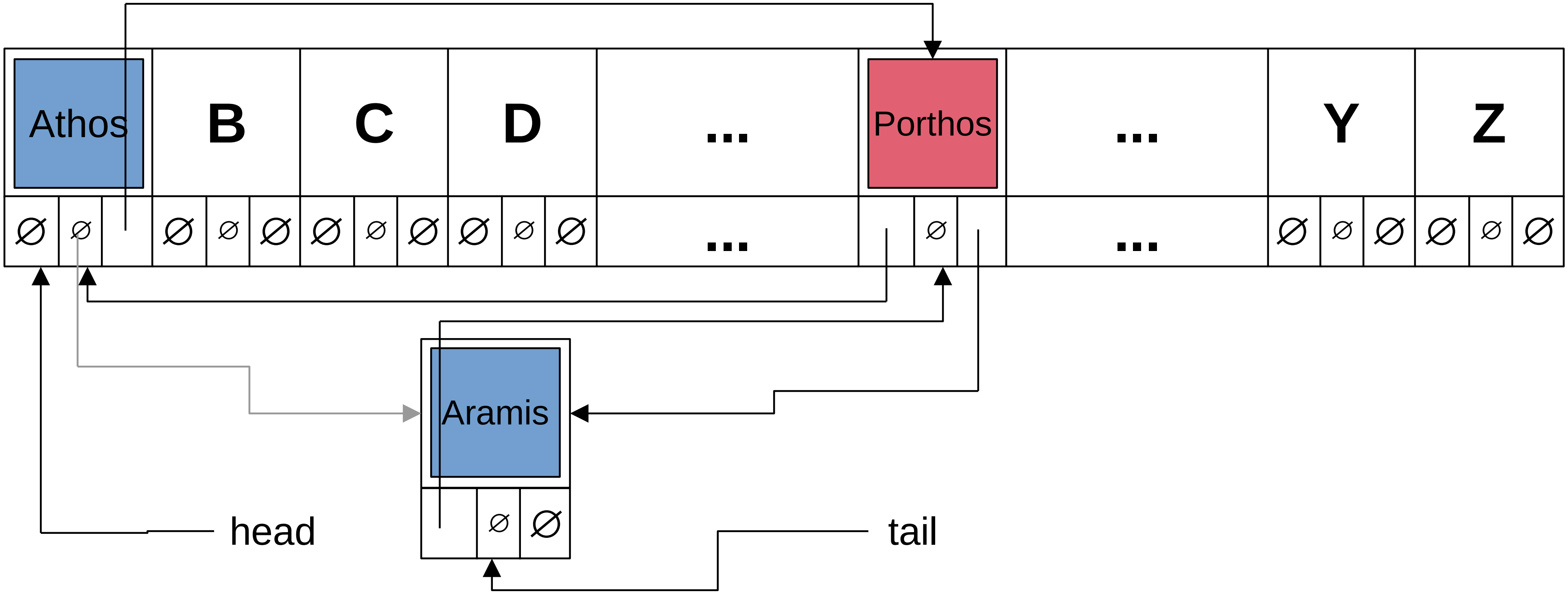
- Use two hash functions: $hash_1$, $hash_2$
  - Each record is stored at one of the two
- insert(x)
  - If both buckets are available: pick at random
  - If one bucket is available: insert record there
  - If neither bucket is available, pick one at random
    - “Displace” the record there, move it to the other bucket
    - Repeat displacement until an empty bucket is found

**apply(x) and remove(x) is guaranteed O(1)**

# Linked Hash Table

- Iteration over Hash Table is O(N + n)
  - Can be much slower than O(n)
- **Idea**: Connect entries together in a Doubly Linked List

# Linked Hash Table



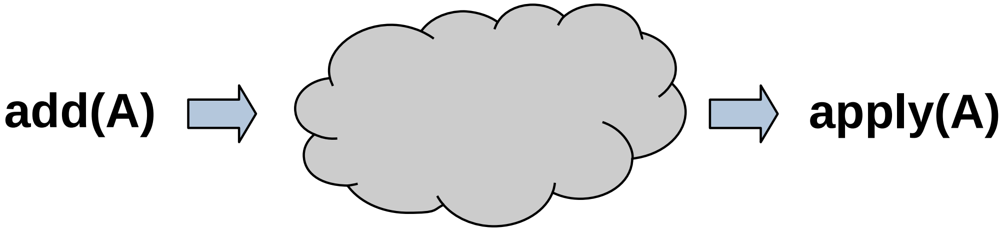
# Linked Hash Table

- O(n) Iteration
- apply(x)
  - O(1) increase in cost
- insert(x)
  - O(1) increase in cost
- remove(x)
  - O(1) increase in cost

# Lossy Sets / Bloom Filters

# “Lossy Sets”

- Set[A]
  - **add(a: A)**: Insert **a** into the set
  - **apply(a: A)**: Return true if **a** is in the set




- What if we didn’t need apply to be perfect?

# Lossy Sets

- LossySet[A]
  - **add(a: A)**: Insert **a** into the set.
  - **apply(a: A)**:
    - If **a** is in the set, <u>always</u> return true
    - If **a** is not in the set, <u>usually</u> return false
      - Is allowed to return true, even if **a** is not in the set

# Bloom Filters

```
class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = {
    for(i <- 0 until _k) {
      if( !bits( ithHash(a, i) % _size ) { return false; }
    }
    return true
  }
}
```

# Bloom Filter Parameters

- _size
  - Intuitively: More space, fewer collisions
- _k
  - Intuitively: more hash functions means...
    - ...more chances for one of **b**'s bits to be unset.
    - ...more bits set = higher chance of collisions.

**To preserve a constant false-positive rate:**
**Grow _size as O(n)**
**Value of _k is fixed for a given size.**

# Aggregation, Joins

# Usage Pattern 1: Aggregation

- Examples:
  - “sum up __, for each__”
  - “average __, by __”
  - “number of __, for __”
  - “biggest __, for each __”
- Pattern
  - (Optionally) Group records by a “Group By” key
  - For each group, compute a statistic
    - e.g., sum, count, average, min, max

# Usage Pattern 1: Aggregation

```
def countBy[A, K](elements: Iterable[A], getKey: A => K): Map[K, Int] =
{
  val result = mutable.Map[K, Int]()
  for(element <- elements){
    val key = getKey(element)
    if(result.contains(key)){
      result(key) += 1
    } else {
      result(key) = 1
    }
  }
  return result
}
```

# Usage Pattern 2: Joins

- Examples:
  - “combine these datasets”
  - “look up __ for each __”
  - “join __ and __ on __”
- Pattern
  - For each record in one dataset...
  - ... find the corresponding record(s) in the other set
  - Output each pair of matched records

# Usage Pattern 2: Joins

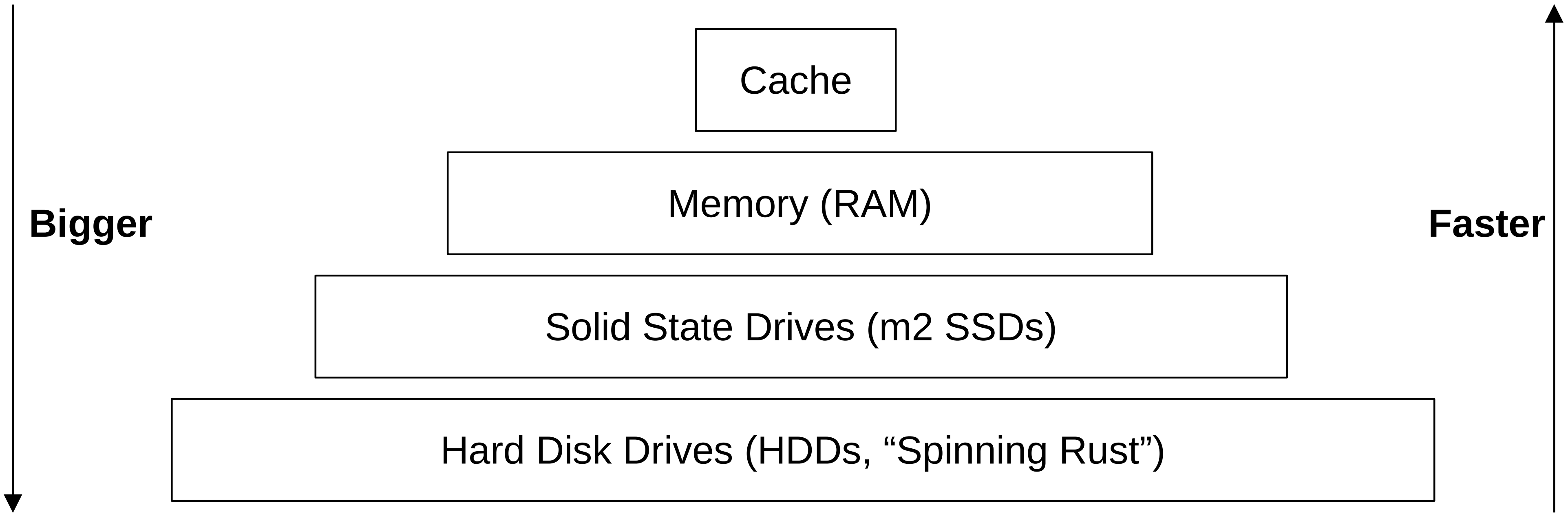
```scala
def nestedLoopJoin(
  sales: Seq[SaleRecord], prices: Seq[ProductPrice]
): mutable.Buffer[(SaleRecord, ProductPrice)] =
{
  val result = mutable.Buffer[(SaleRecord, ProductPrice)]()
  for(s <- sales){
    for(p <- prices){
      if(s.productId == p.productId){
        result += ( (s, p) )
      }
    }
  }
  return result
}
```

# Usage Pattern 2: Joins

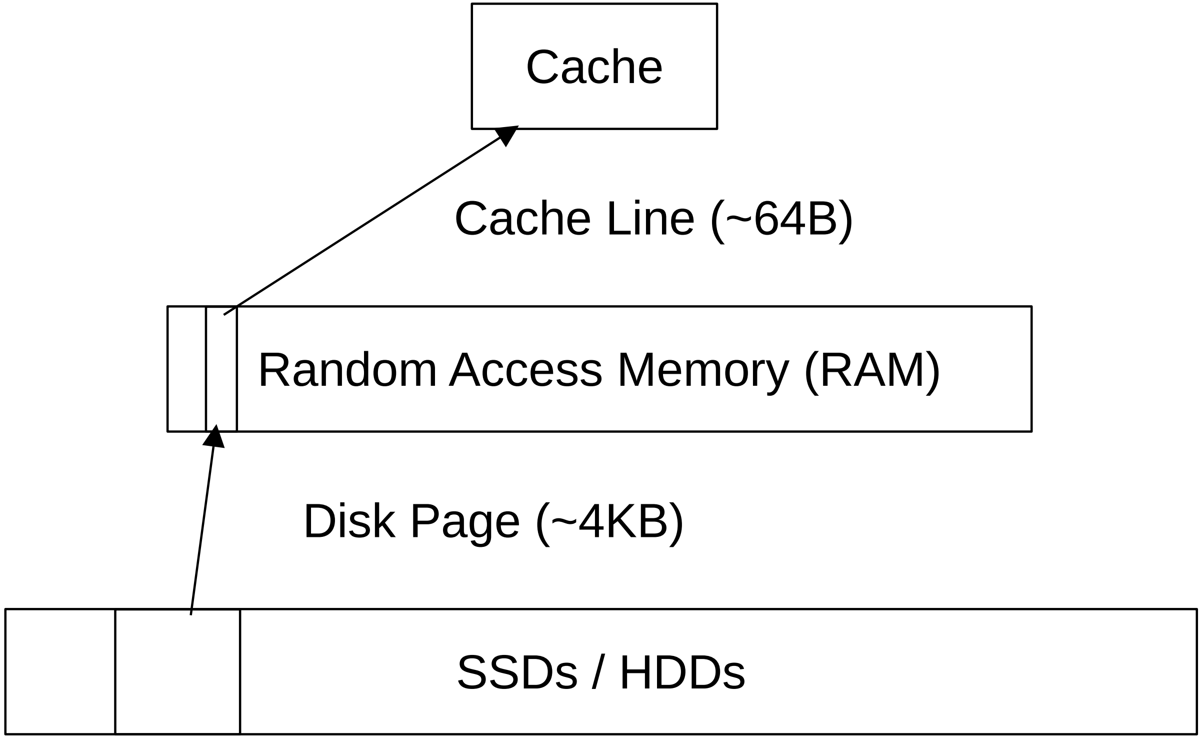
```
def hashJoin(
  sales: Seq[SaleRecord], prices: Seq[ProductPrice]
): mutable.Buffer[(SaleRecord, ProductPrice)] =
{
  val indexedPrices = mutable.HashMap[Int, ProductPrice]()
  for(p <- prices){
    indexedPrices(p.productId) = p
  }
  val result = mutable.Buffer[(SaleRecord, ProductPrice)]()
  for(s <- sales){
    if(indexedPrices.contains(s.productId)){
      result += ( (s, indexedPrices(s.productId)) )
    }
  }
  return result
}
```

# Memory Hierarchy

# The Memory Hierarchy (simplified)



# The Memory Hierarchy (simplified)



# Reading an Array Entry

- Is the array entry in cache?
  - Yes
    - Return it (1-4 clock cycles)
  - No
    - Is the array entry in real memory
      - Yes
        - Load it into cache (10s of clock cycles)
      - No
        - Load it out of virtual memory (100s of clock cycles)

Tiny constant

So-so constant

HUGE constant

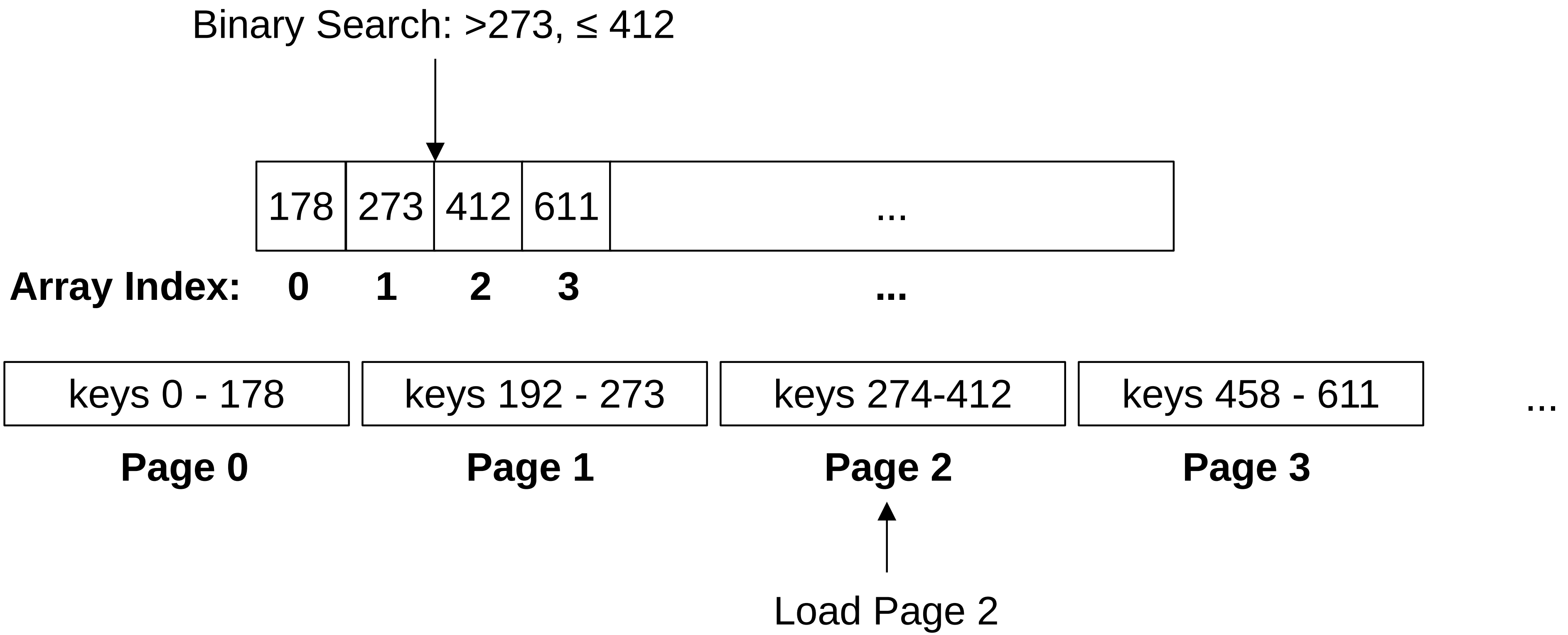
# Fence Pointers

- **Idea**: Precompute the greatest key in each page in memory
  - n records; 64 records/page; $^{n}/_{64}$ keys
  - e.g., $n=2^{20}$ records; Needs $2^{14}$ keys
    - $2^{20}$ 64 byte records = 64 MB
    - $2^{14}$ 8 byte records = $2^{19}$ bytes = 512 ***K***B
  - Call this a "Fence Pointer Table"

**RAM:** | $2^{14}$ = 16,384 keys (Fence Pointer Table)

**Disk:** | 16,384 pages (Actual Data)

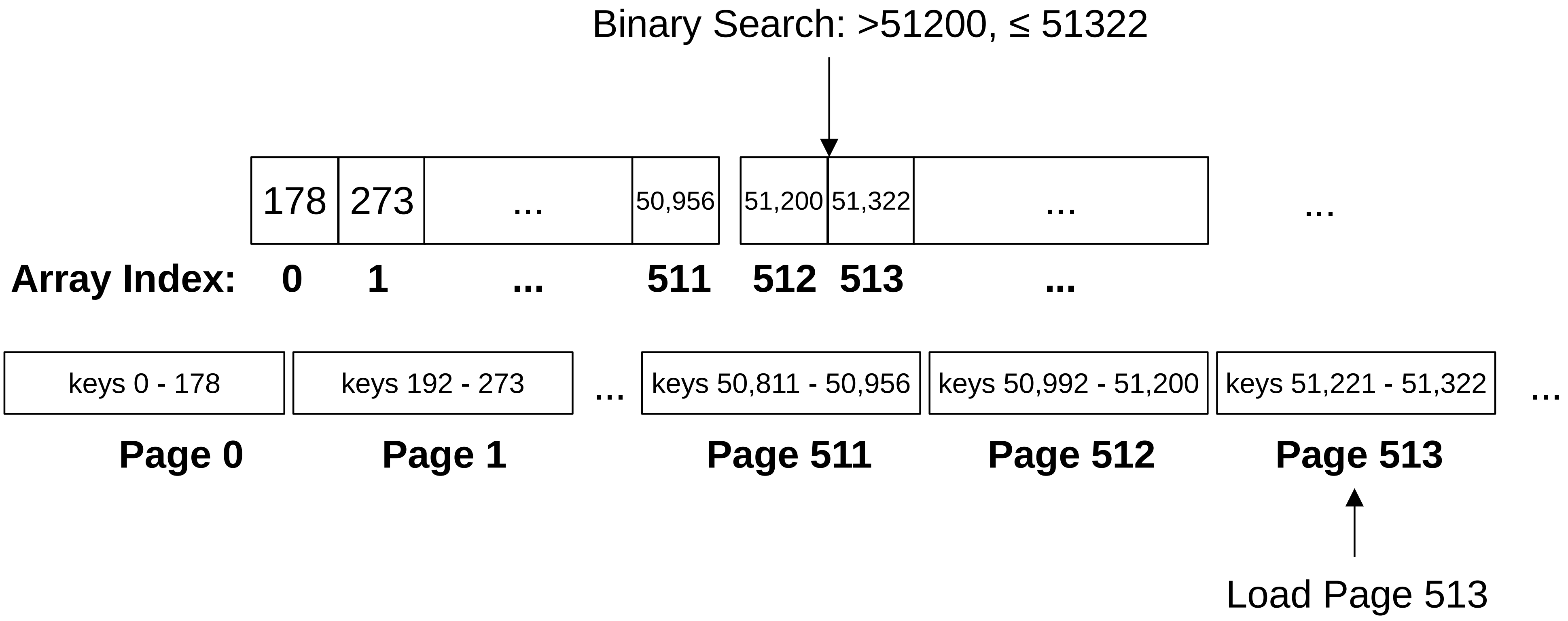
# Example


Binary Search: >273, ≤ 412
178
273
412
611
...
Array Index:
0
1
2
3
...
keys 0 - 178
keys 192 - 273
keys 274-412
keys 458 - 611
...
Page 0
Page 1
Page 2
Page 3
Load Page 2

# Fence Pointers

- Memory Complexity:
  - Need the entire fence pointer table in memory **at all times**
    - O(n / C) pages = O(n)
  - Steps 2, 3 load one more page
  - **Total**: O(n+1) = O(n)

# Example



Binary Search: >51200, ≤ 51322
178
273
...
50,956
51,200
51,322
...
...
Array Index:
0
1
...
511
512
513
...
keys 0 - 178
keys 192 - 273
...
keys 50,811 - 50,956
keys 50,992 - 51,200
keys 51,221 - 51,322
...
Page 0
Page 1
Page 511
Page 512
Page 513
Load Page 513

# Improving on Fence Pointers

- **Idea**: Multiple levels of fence pointers
  - Store the greatest key of each fence pointer page.
  - If it fits in memory, done!
  - If not, add another level

# ISAM Index

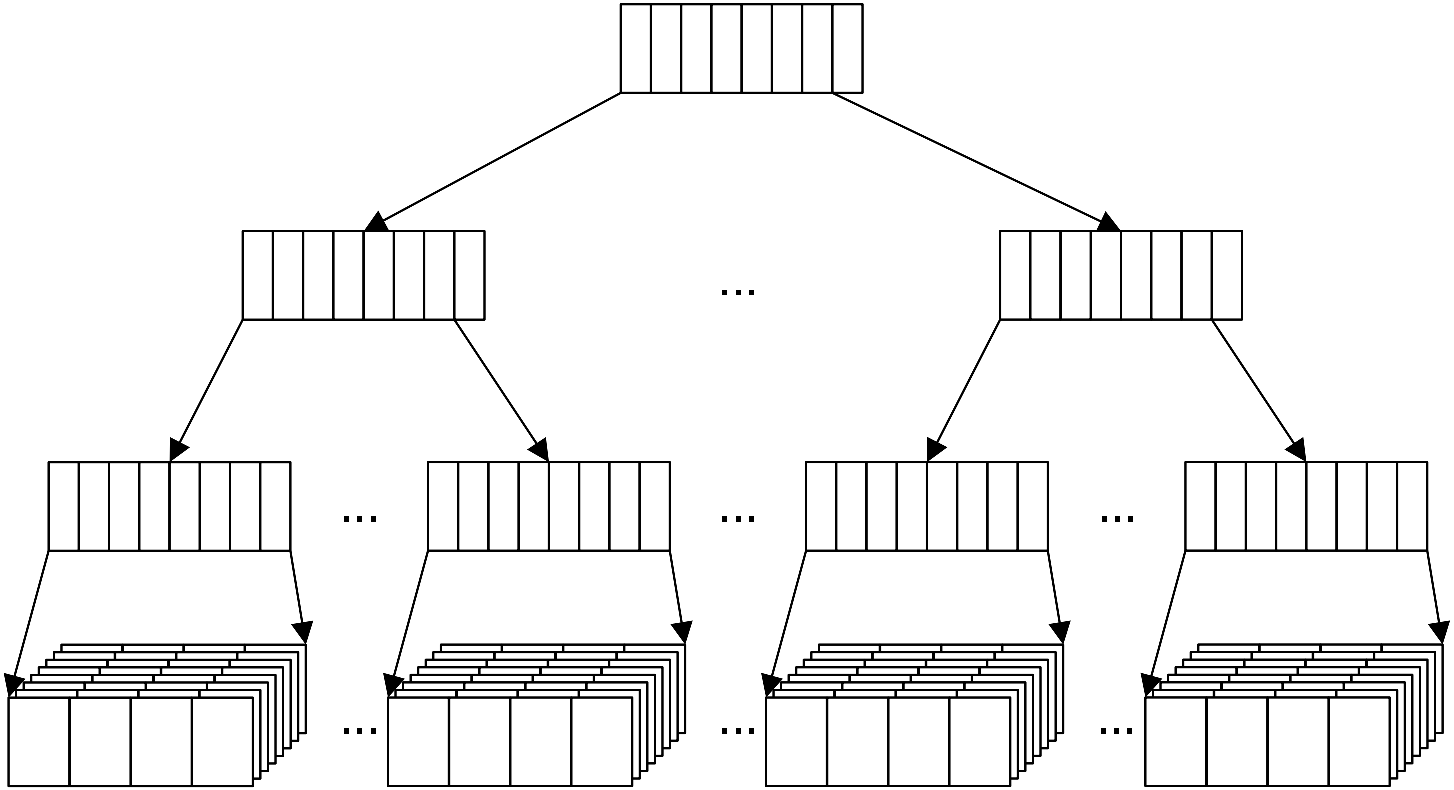
# ~~Improving on Fence Pointers~~

Binary Search @ Level 0
to find a Level 1 page

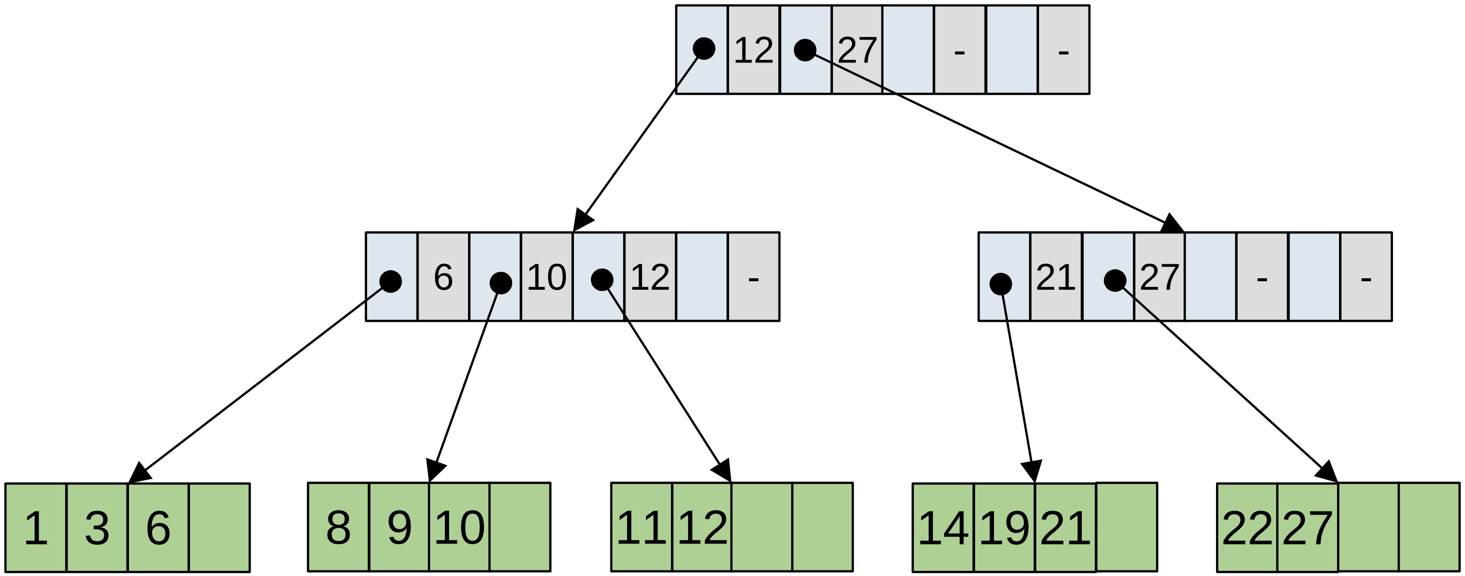
Binary Search @ Level 1
to find a Level 2 page

Binary Search @ Level 2
to find a Data page

Binary Search @ Data
to find the record



# B+ Trees



# B+ Trees

- **Observation**: Don’t need the biggest key
- **Question**: What if the separator value is mispositioned?
  - **Idea:** “Steal” space from adjacent nodes
- **Question**: What happens when we delete records?
  - **Observation**: The tree becomes unbalanced
    - **Idea**: “Minimum Fill”

# B+ Trees

- **Insert**:
  - Find the page that the record belongs on
  - Insert record there
  - If full, “split” the page
    - Insert additional separator in parent directory page
    - If full, “split” the directory page and repeat with parent
      - If “root split” create a new parent node

# B+ Trees

- **Delete**:
  - Find the page that the record is on
  - Delete record (if present)
  - If underfull, "merge" the page with a neighbor
    - If either neighbor at > $^{c}/_{2}$ entries (can't merge)
      - "steal" entries from neighbor
    - If parent underfull, "merge" parent with neighbor
      - Repeat as needed
      - If "root merge" drop lowest layer

# Spatial Indexes

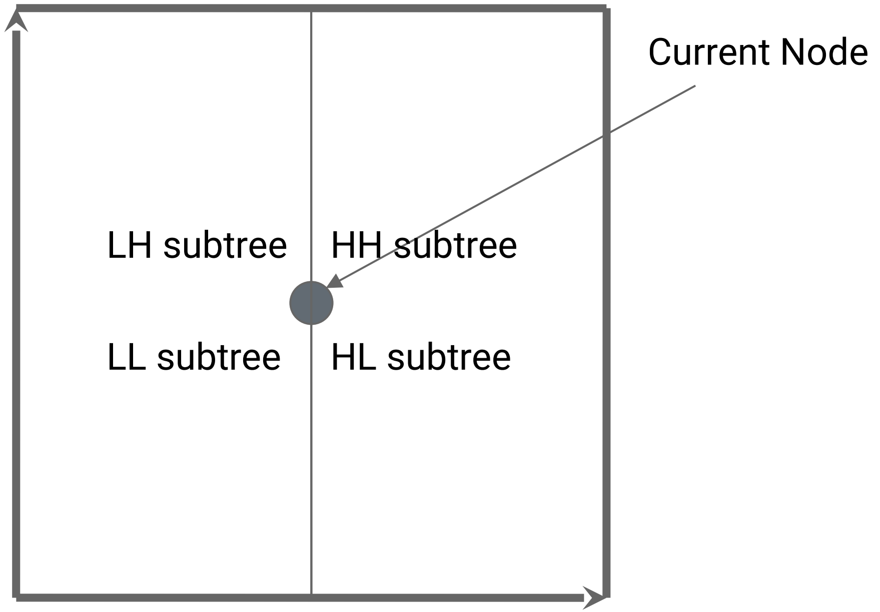
# The 2D Map ADT

2DMap[T]

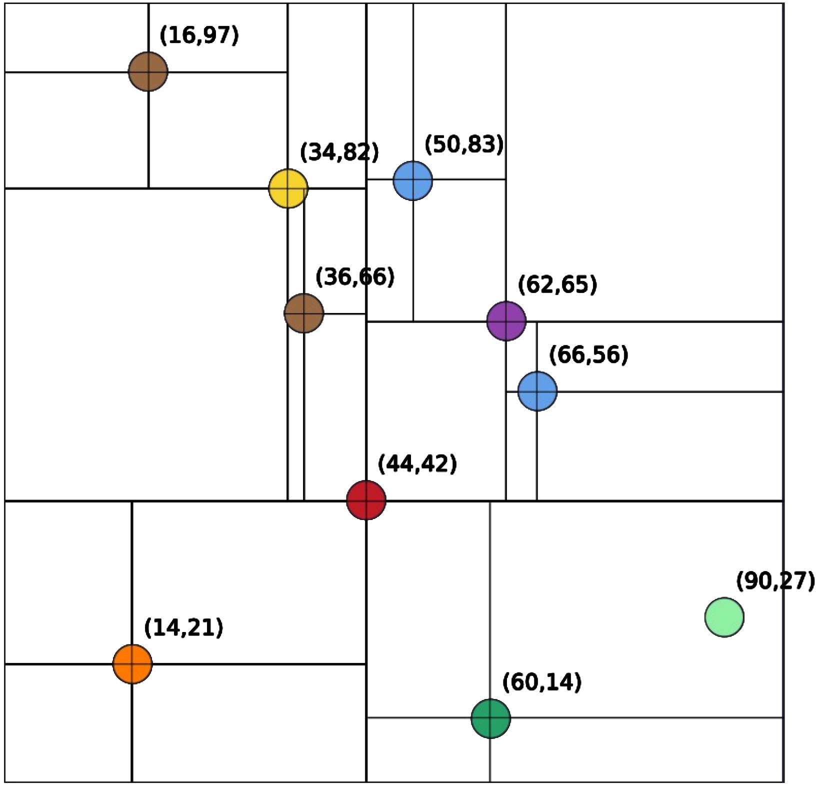
- insert(x: Int, y: Int, value: T): Unit
    - Add an element to the map at point **(x, y)**
- apply(x: Int, y: Int): T
    - Retrieve the element at point **(x, y)**
- range(xlow: Int, xhigh: Int, ylow: Int, yhigh: Int): Iterator[T]
    - Retrieve all elements in the rectangle defined by **( [xlow, xhigh), [ylow, yhigh) )**
- knn(x: Int, y: Int, k: Int)
    - Retrieve the k elements closest to the point **(x, y)** (k-nearest neighbor)

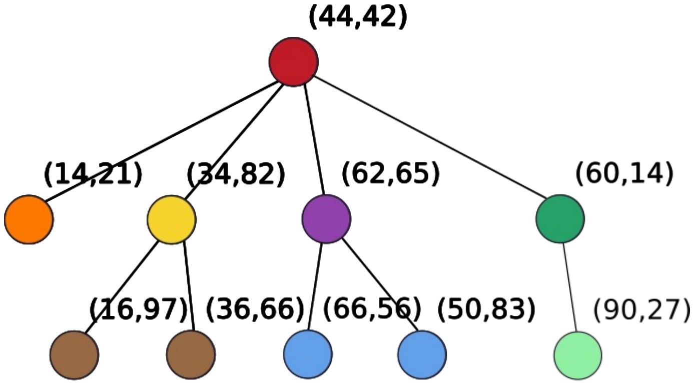
# Attempt 1: Quad Trees

Possible Values:



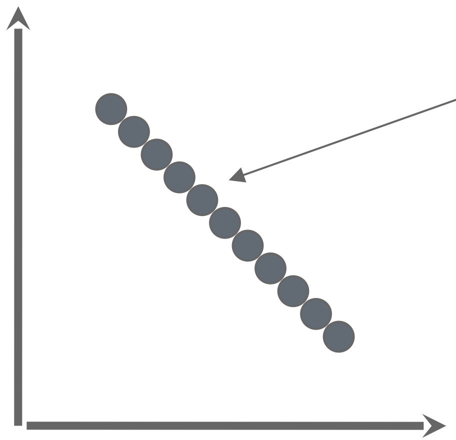
# Each Node has 4 Children



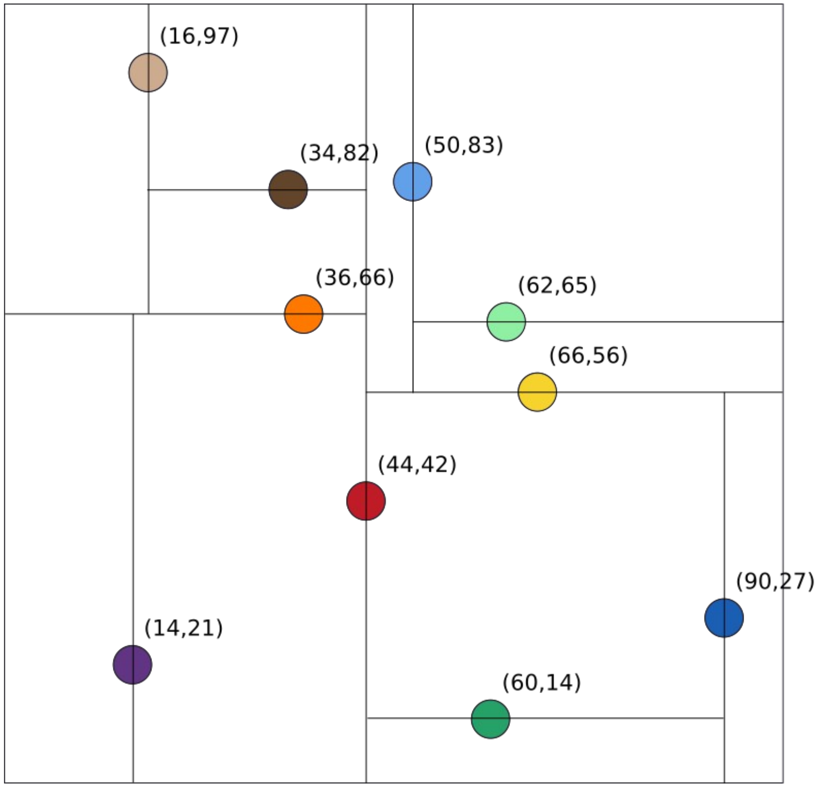
# Quad Tree: Challenges

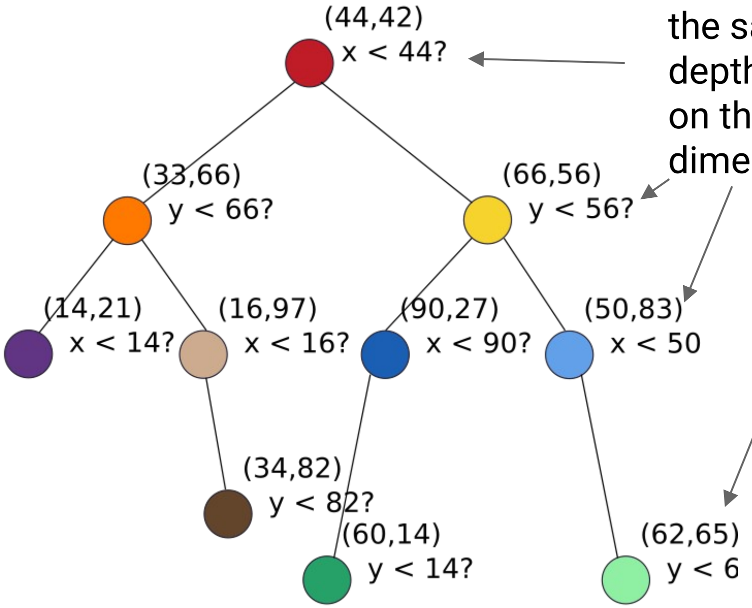
- Creating a balanced Quad Tree is hard
  - Impossible to always split collection elements evenly across all four subtrees (though depth = O(log(n)) still possible)
- Keeping the quad tree balanced after updates is significantly harder
  - No “simple” analog for rotate left/right.



**Worst Case:**
No possible way to create nodes with >2 nonempty subtrees

# k-D Trees




All nodes at the same depth partition on the same dimension

# Wrap-Up

# TA Positions

- Did you enjoy what you learned here and want to share it with others?
- Did you hate what you learned here and think you can teach it better?
- Do you feel like you want to learn the material even better?

- Be a TA!
  - email me <okennedy@buffalo.edu>

# Research

- Using data structures to make compilers faster
  - https://github.com/UBOdin/jitd-synthesis
- Interactive tools for data exploration/visualization
  - https://vizierdb.info
- Collaborations w/ Materials Science, Food Systems
  - Websites in progress
- Managing ambiguity, corner cases, and wackiness in data
  - https://mimirdb.info

# Thanks for a great semester!