CISC 235: Topic 8

Internal and External Sorting

External Searching

Outline

- Internal Sorting
  - Heapsort
- External Sorting
  - Multiway Merge
- External Searching
  - B-Trees

Heapsort

Idea: Use a max heap in a sorting algorithm to sort an array into increasing order.

Heapsort Steps

1. Build a max heap from an unsorted array
2. Remove the maximum from the heap n times and store in an array

We could keep the heap in one array and copy the maximum to a second array n times.

Heapsort in a Single Array

Heapsort Steps

1. Build a max heap from an unsorted array

2a. Remove the largest from the heap and place it in the last position in array

2b. Remove the 2nd largest from the heap and place it in the 2nd from last position

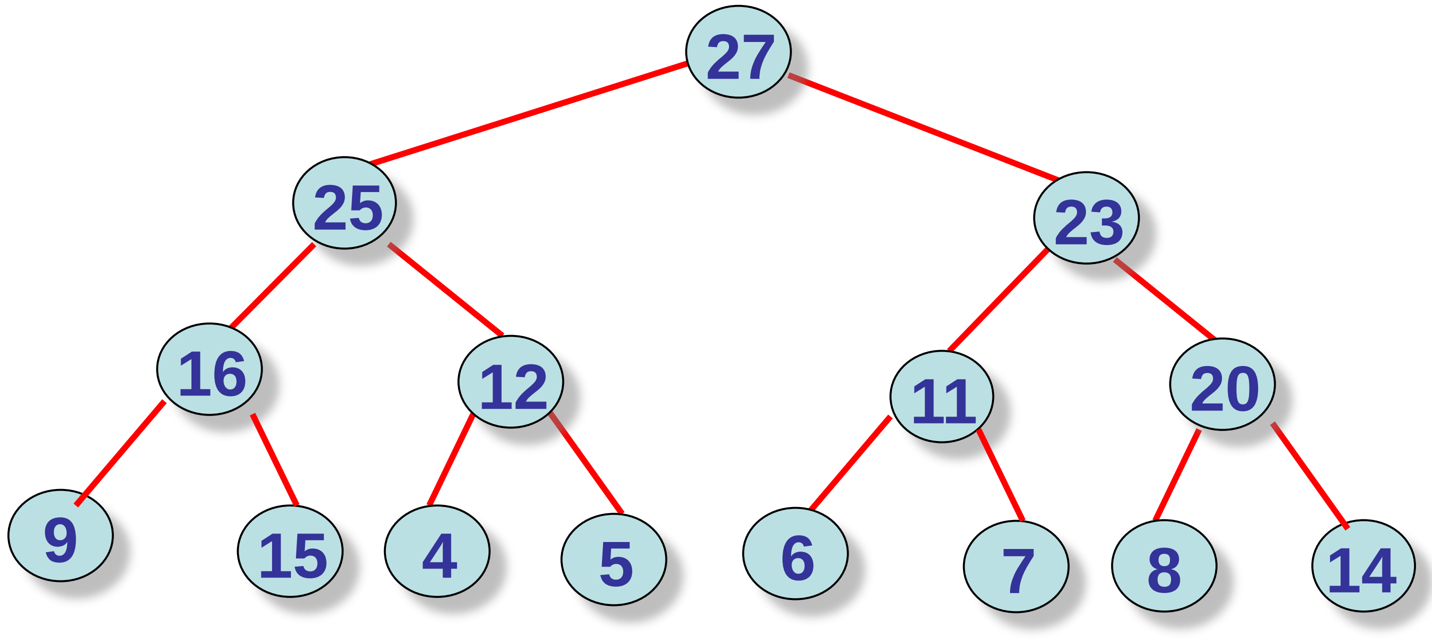
2c. Remove the 3rd largest from the heap and place it in the 3rd from last postion

. . . etc.

Heapsort : Start with an Unsorted Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	9	8	25	5	11	27	16	15	4	12	6	7	23	20

Heapsort Step 1: Build a Max Heap from the Array



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	27	25	23	16	12	11	20	9	15	4	5	6	7	8	14

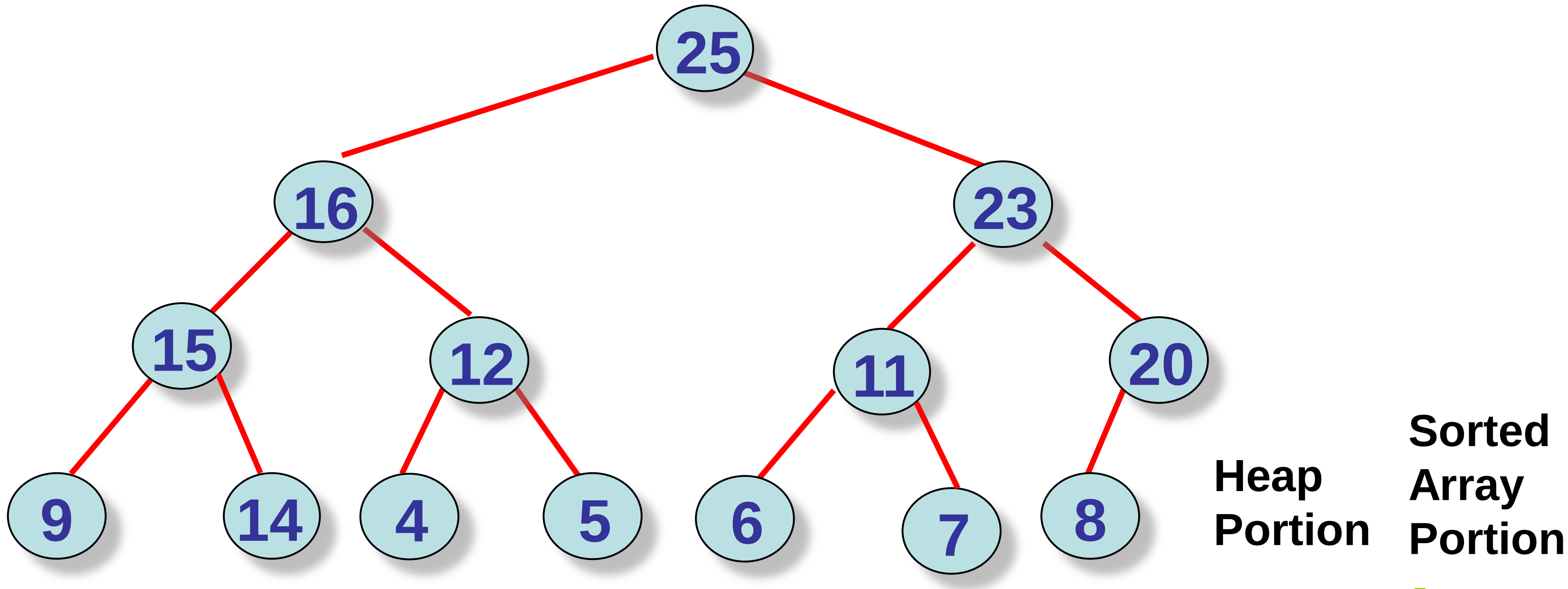
Percolate Down Algorithm for max heap

```
// Heap is represented by array A with two attributes:
//            length[A] and heap-size[A]
// Percolate element at position i down until A[ i ] ≥ its children
Max-Heapify( A, i )
        L ← Left( i )
        R ← Right( i )
        **if**( L ≤ heap-size[A] and A[L] > A[i] )
                **then** largest ← L
                **else** largest ← i
        **if**( R ≤ heap-size[A] and A[R] > A[largest] )
                **then** largest ← R
        **if**( largest ≠ i )
                **then** exchange A[i] ←→ A[largest]
                     Max-Heapify( A, largest )
```

BuildHeap Algorithm for max heap

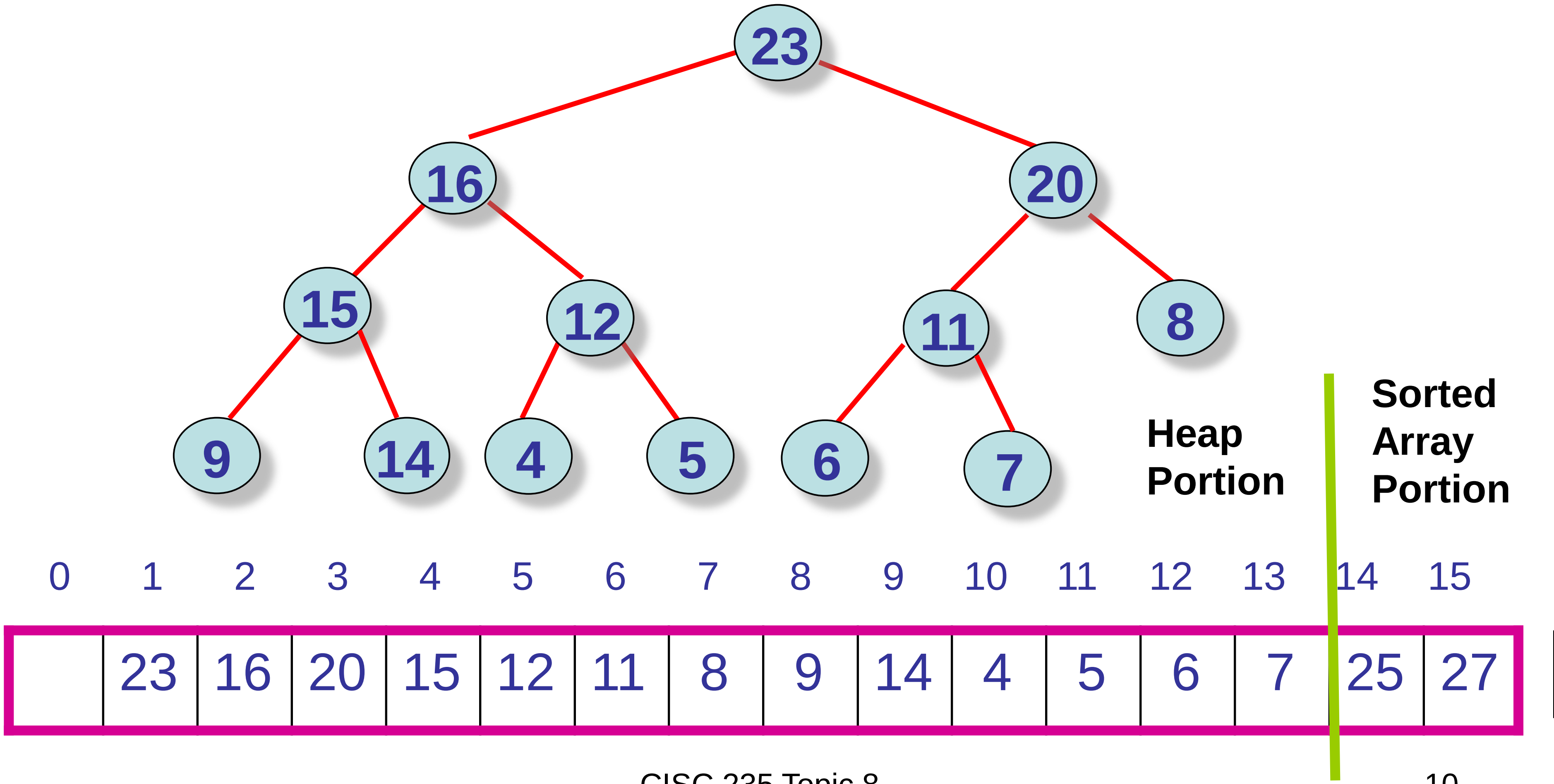
```
// Convert array A to max heap order using
// reverse level-order traversal, calling Max-Heapify
// for each element, starting at the parent of
// the last element in array
Build-Max-Heap( A )
    heap-size[A] ← length[A]
    for i ← ⌊ length[A] / 2 ⌋ down to 1
        do Max-Heapify( A, i )
```

Heapsort Step 2a: Remove largest and place in last position in array

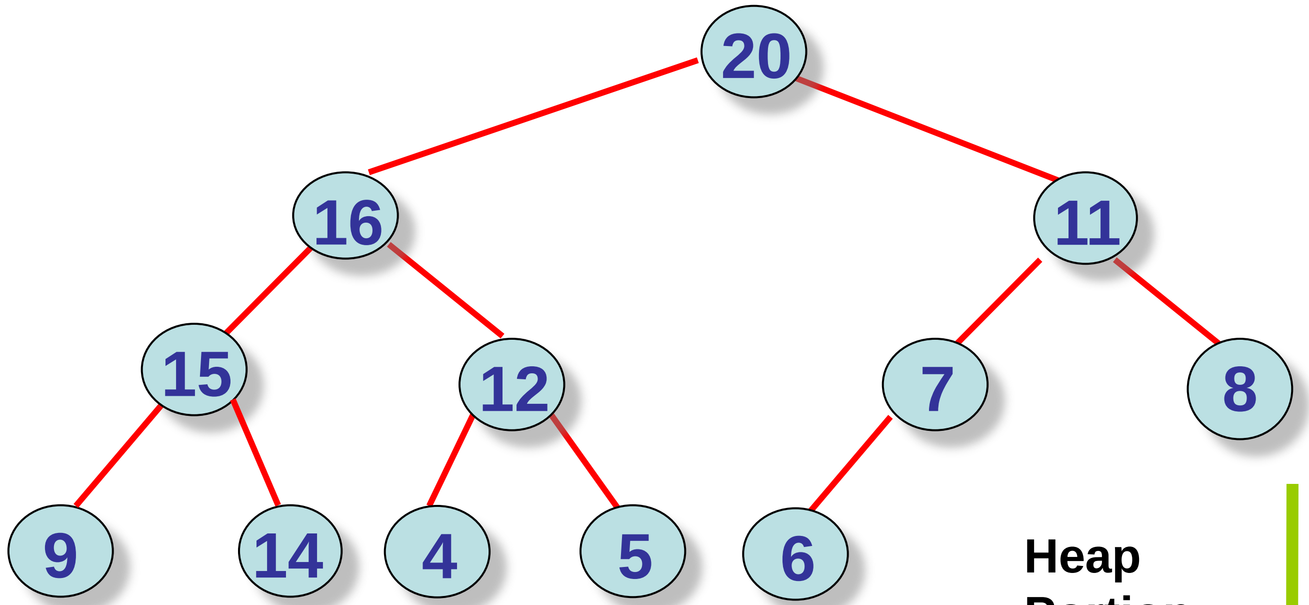


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	25	16	23	15	12	11	20	9	14	4	5	6	7	8	27

Heapsort Step 2b: Remove 2nd largest and place in 2nd from last position in array



Heapsort Step 2c: Remove 3rd largest and place in 3rd from last position in array

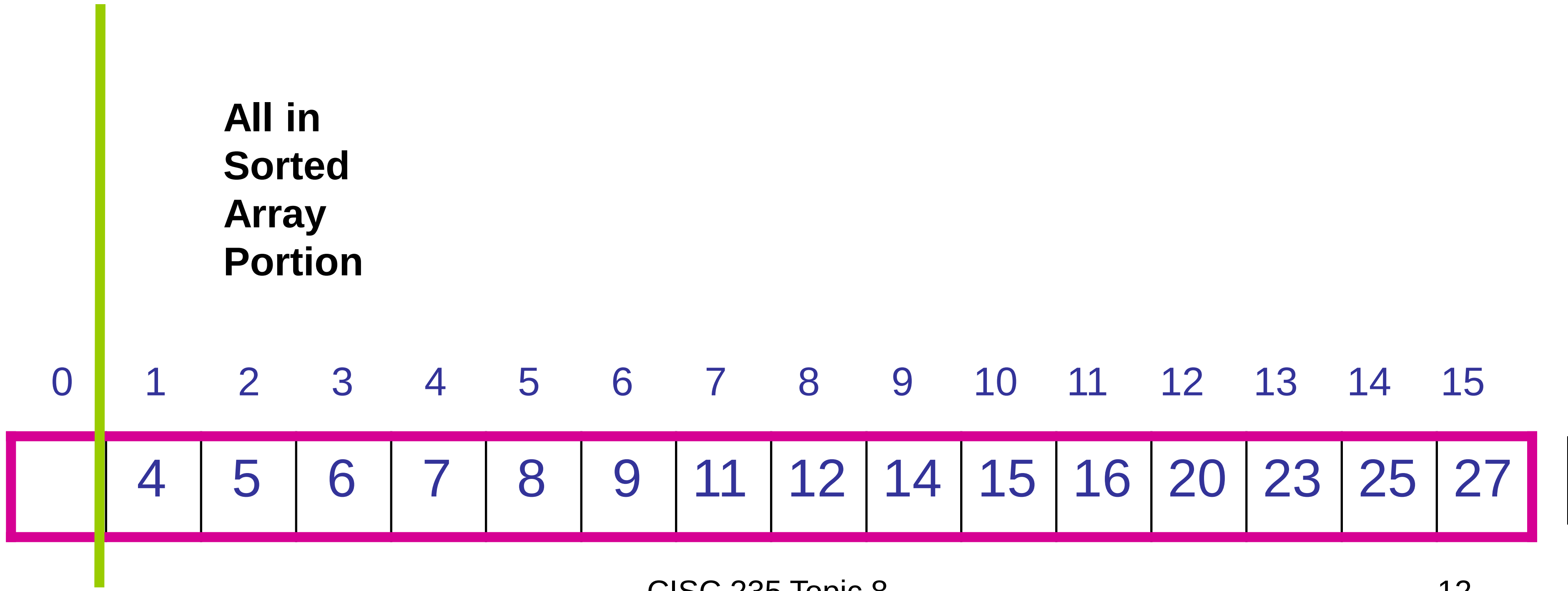


Heap Portion

Sorted Array Portion

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	20	16	11	15	12	7	8	9	14	4	5	6	23	25	27

Heapsort at End: All nodes removed from Heap and now in Sorted Array



Heapsort Analysis?

Worst-case Complexity?

Step 1. Build Heap

Step 2. Remove max n times

Comparison with other good Sorting Algs?

Quicksort?

Mergesort?

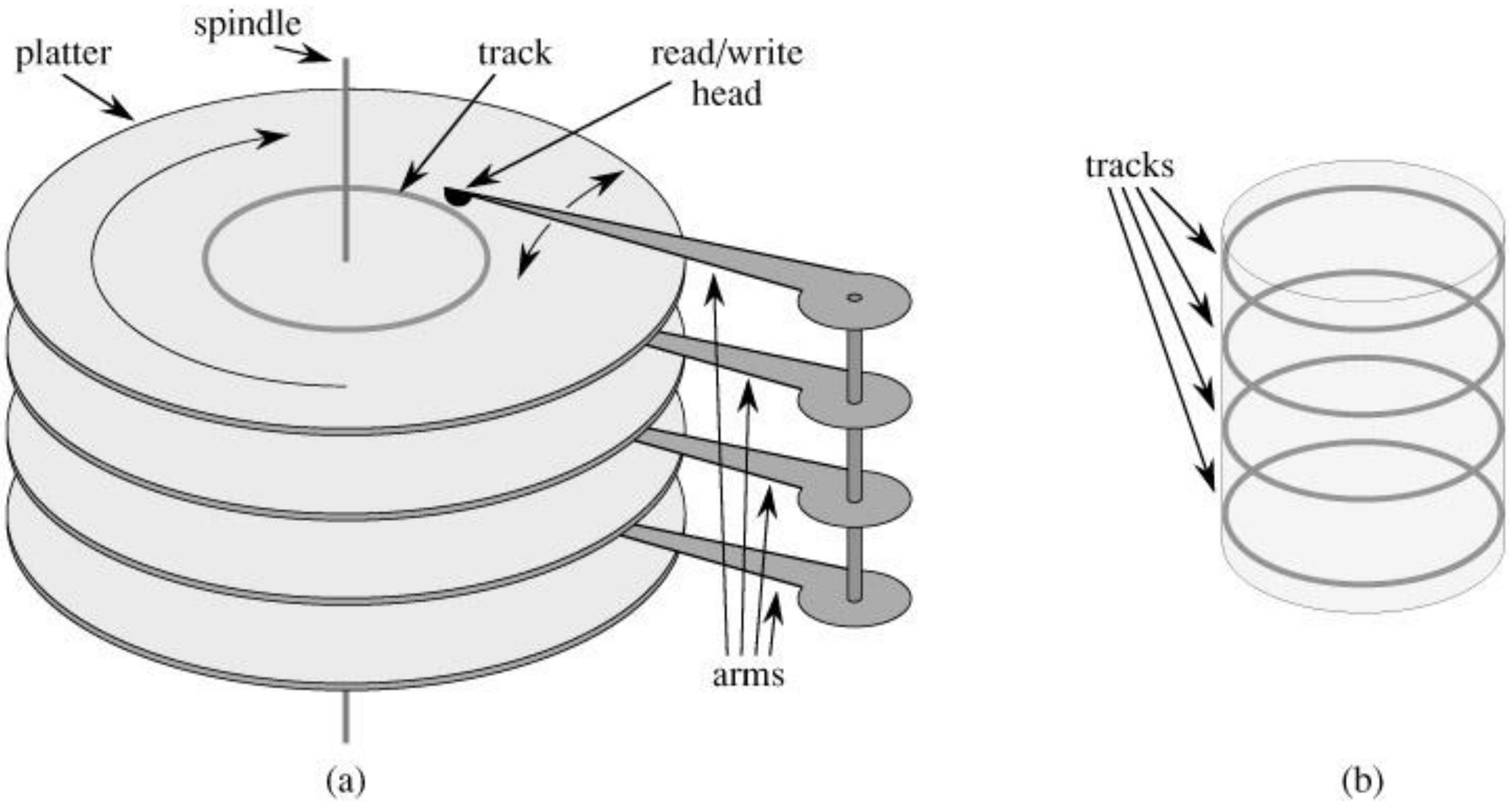
External Sorting

Problem: If a list is too large to fit in main memory, the time required to access a data value on a disk or tape dominates any efficiency analysis.

1 disk access ≡ Several *million* machine instructions

Solution: Develop external sorting algorithms that minimize disk accesses

A Typical Disk Drive



Disk Access

Disk Access Time =

Seek Time (moving disk head to correct track)

+ Rotational Delay (rotating disk to correct block in track)

+ Transfer Time (time to transfer block of data to main memory)

Basic External Sorting Algorithm

- Assume unsorted data is on disk at start
- Let M = maximum number of records that can be stored & sorted in internal memory at one time

Algorithm

Repeat:

1. Read M records into main memory & sort internally.

2. Write this sorted sub-list onto disk. (This is one “run”).

Until all data is processed into runs

Repeat:

1. Merge two runs into one sorted run twice as long

2. Write this single run back onto disk

Until all runs processed into runs twice as long

Merge runs again as often as needed until only one large run: the sorted list

Basic External Sorting

81 94 11 96 12 35 17 99 28 58 41 75 15

Unsorted Data on Disk

Assume M = 3 (M would actually be much larger, of course.) First step is to read 3 data items at a time into main memory, sort them and write them back to disk as runs of length 3.

11 81 94   17 28 99   15

12 35 96   41 58 75

Basic External Sorting

11 81 94   17 28 99   15

12 35 96   41 58 75

Next step is to merge the runs of length 3 into runs of length 6.

11 12 35 81 94 96

17 28 41 58 75 99

15

Basic External Sorting

11 12 35 81 94 96   15

17 28 41 58 75 99

Next step is to merge the runs of length 6 into runs of length 12.

11 12 17 28 35 41 58 75 81 94 96 99

15

Basic External Sorting

11	12	17	28	35	41	58	75	81	94	96	99

15

Next step is to merge the runs of length 12 into runs of length 24. Here we have less than 24, so we're finished.

11	12	15	17	28	35	41	58	75	81	94	96	99

Multi-way Mergesort

Idea: Do a K-way merge instead of a 2-way merge.

Find the smallest of K elements at each merge step. Can use a priority queue internally, implemented as a heap.

Multi-way Mergesort Algorithm

Algorithm:

1. As before, read M values at a time into internal memory, sort, and write as runs on disk
2. Merge K runs:
    1. **Read first value on each of the k runs into internal array and build min heap**
    2. **Remove minimum from heap and write to disk**
    3. **Read next value from disk and insert that value on heap**

    Repeat steps until all first K runs are processed

- Repeat merge on larger & larger runs until have just one large run: sorted list

Multi-way Mergesort Analysis

Let N = Number of records

B = Size of a Block (in records)

M = Size of internal memory (in records)

K = Number of runs to merge at once

Simplifying Assumptions: N & M are an exact number of blocks (no part blocks):

$N = c_n B$, a constant times B

$M = c_m B$, a constant times B

Multi-way Mergesort Analysis

Specific Example:

M = 80 records

B = 10 records

N = 16,000,000 records

So, $K = \frac{1}{2}(M/B) = \frac{1}{2}(80/10) = 4$

Multi-way Mergesort Analysis: Advantage Gained with Heap

External Searching

Problem: We need to maintain a sorted list to facilitate searching, with insertions and deletions occurring, but we have more data than can fit in main memory.

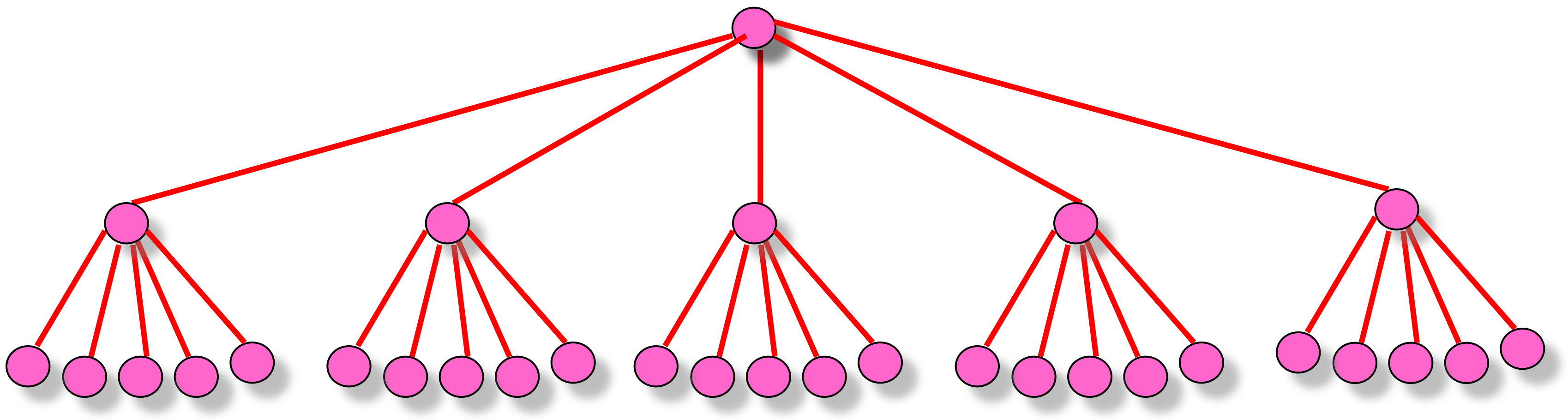
Task: Design a data structure that will minimize disk accesses.

Idea: Instead of a binary tree, use a balanced M-ary tree to reduce levels and thus reduce disk accesses during searches. Also, keep many keys in each node, instead of only one.

M-ary Tree

A 5-ary tree of 31 nodes has only 3 levels.

Note that each node in a binary tree could be at a different place on disk, so we have to assume that following any branch (edge) is a disk access. So, minimizing the number of levels minimizes the disk accesses.

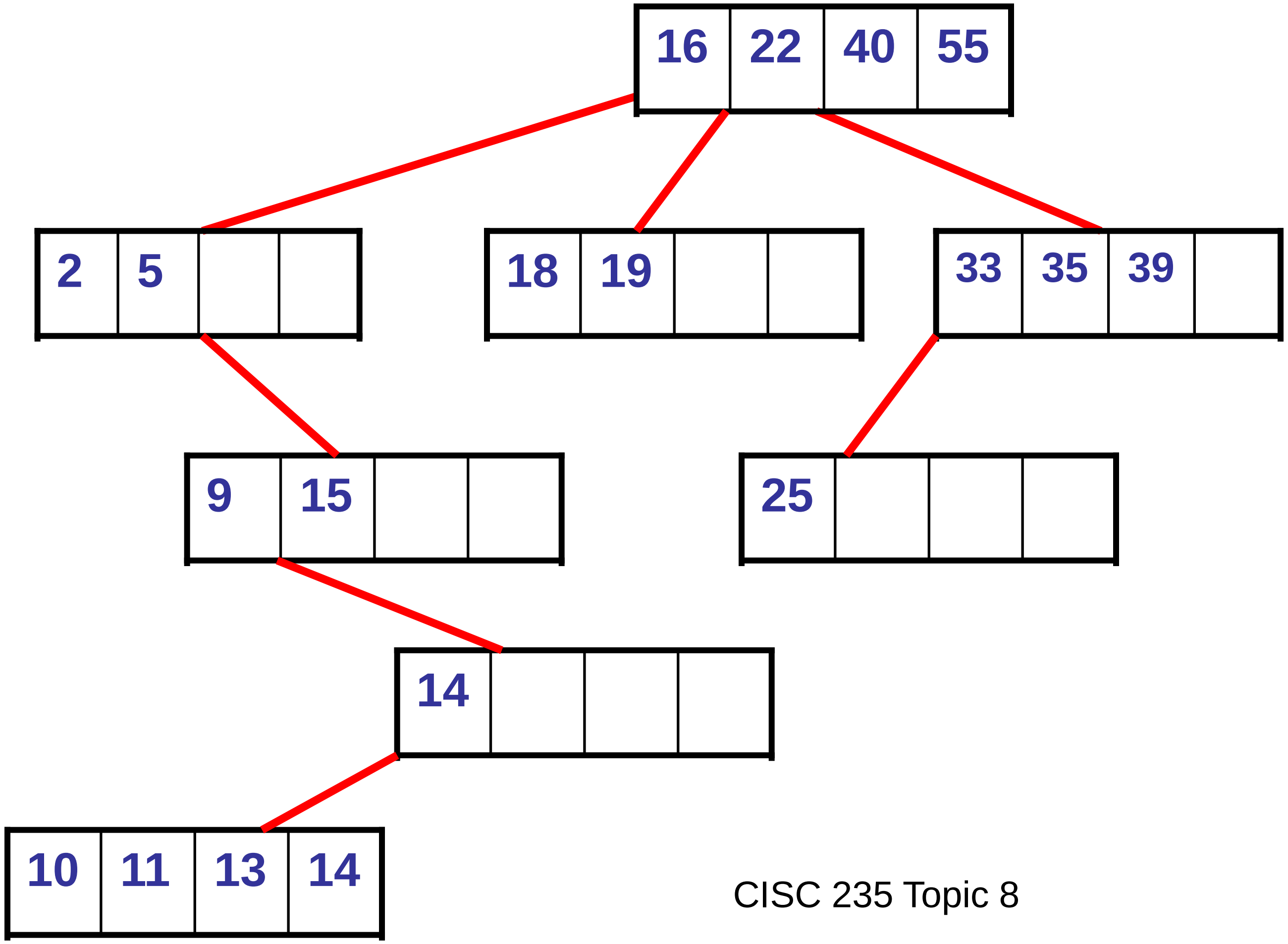


Multiway Search Trees

A *multiway search tree* of order m, or an *m-way search tree*, is an m-ary tree in which:

1. Each node has up to m children and m-1 keys
2. The keys in each node are in ascending order
3. The keys in the first i children are smaller than the ith key
4. The keys in the last m-i children are larger than the ith key

A 5-Way Search Tree



B-Trees

A *B-Tree* is an m-Way search tree that is always at least half-full and is perfectly balanced.

A B-Tree of order m has the properties:

1. The root has at least two sub-trees, unless it's a leaf
2. Each non-root and non-leaf node holds k-1 keys and k pointers to sub-trees, where

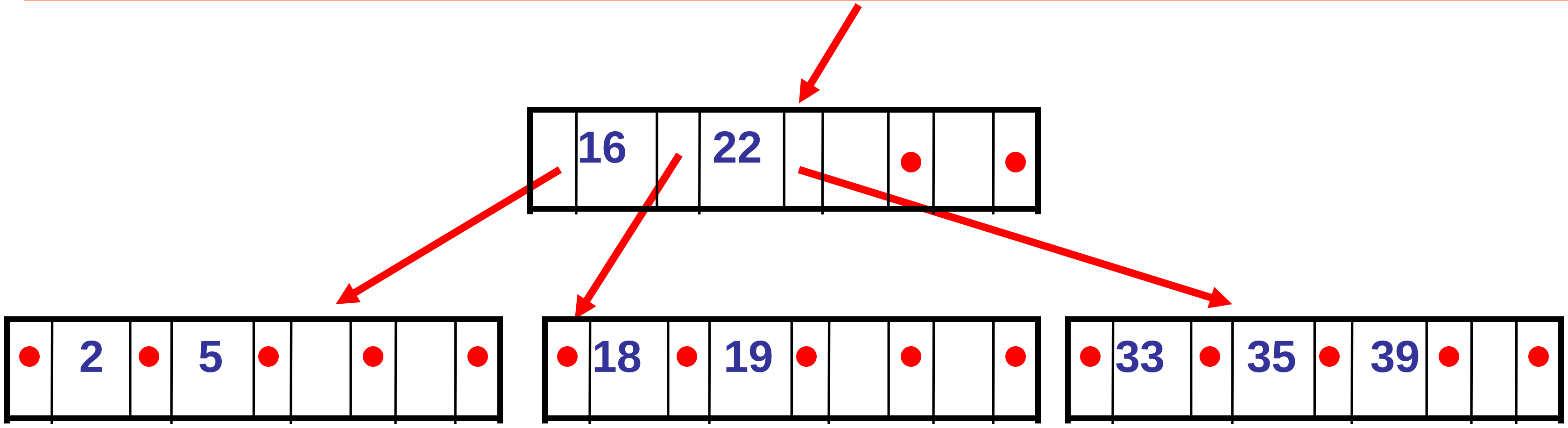
   $\lceil m/2 \rceil \leq k \leq m$

   (i.e., internal nodes are at least half-full)
3. Each leaf node holds k-1 keys, where

   $\lceil m/2 \rceil \leq k \leq m$

   (i.e., leaf nodes are at least half-full)
4. All leaves are on the same level.

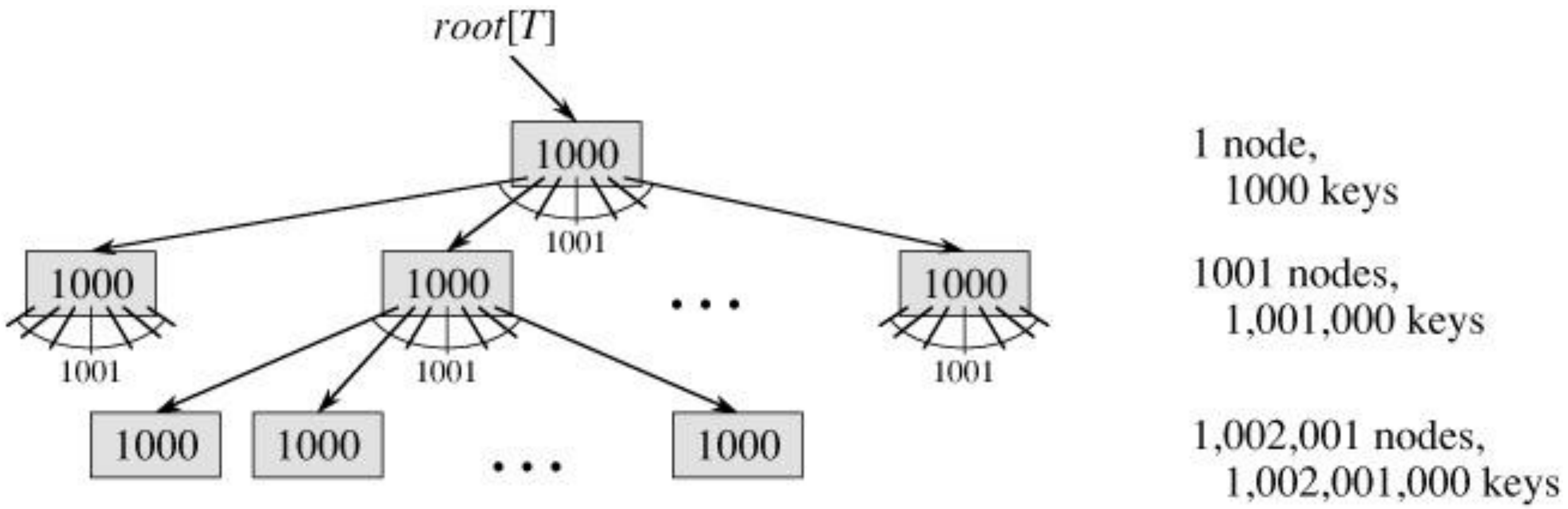
A B-Tree of Order 5



To find the location of a key, traverse the keys at the root sequentially until at a pointer where any key before it is less than the search key and any key after it is greater than or equal to the search key.

Follow that pointer and proceed in the same way with the keys at that node until the search key is found, or are at a leaf and the search key is not in the leaf.

A B-Tree of Order 1001



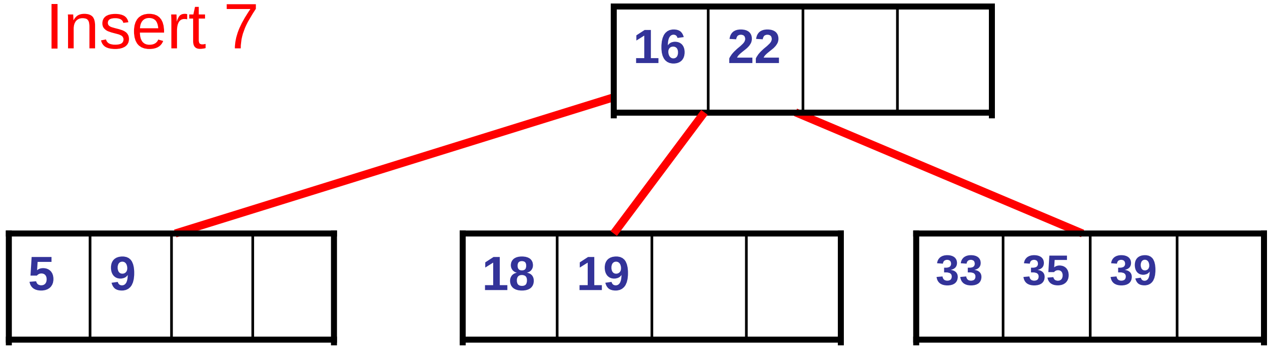
2-3-4 Trees

In a B-Tree of what order will each internal node have 2, 3, or 4 children?

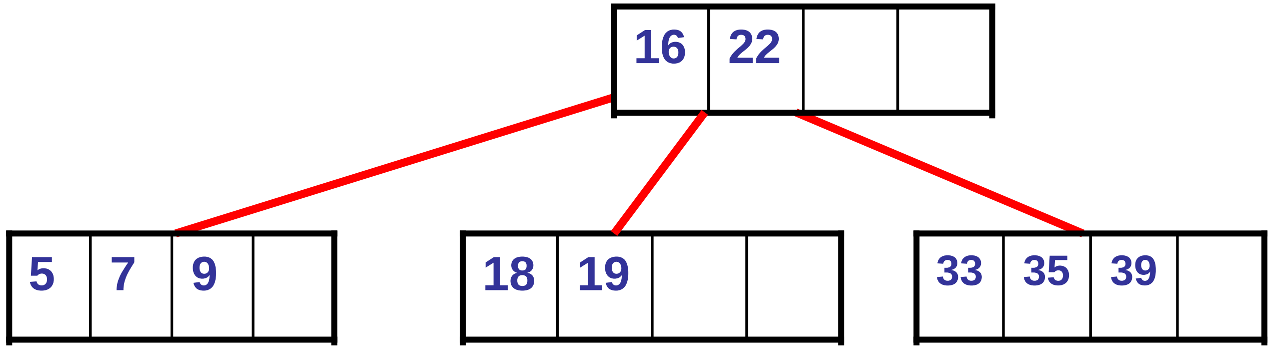
B-Tree Insertion Case 1:

A key is placed in a leaf that still has some room

Insert 7



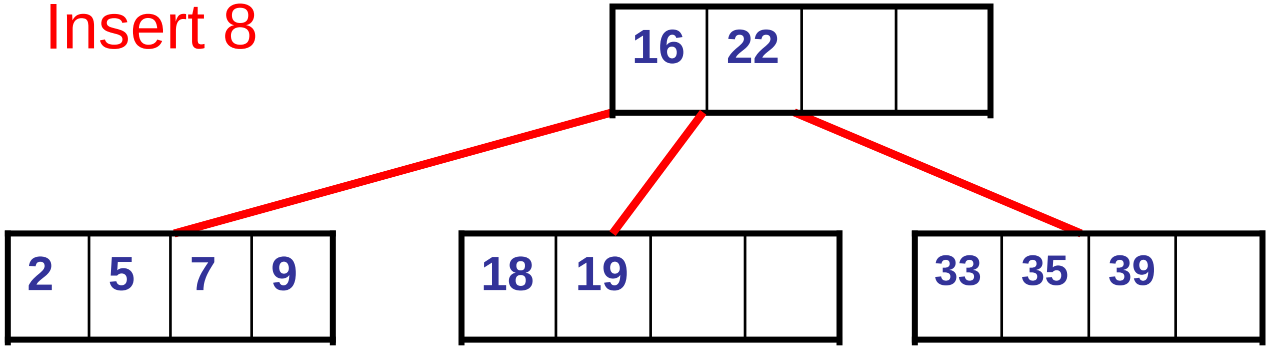
Shift keys to preserve ordering & insert new key.



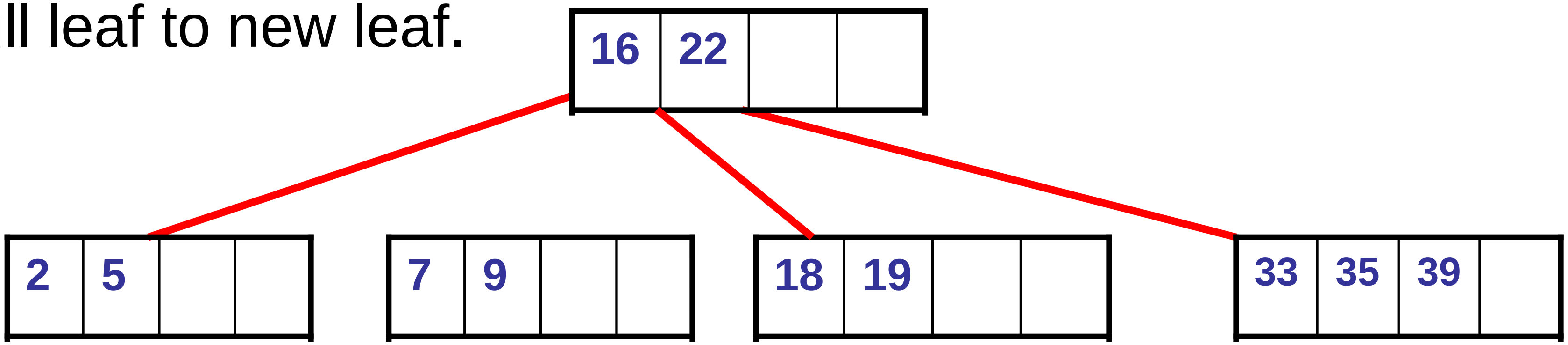
B-Tree Insertion Case 2:

A key is placed in a leaf that is full

Insert 8

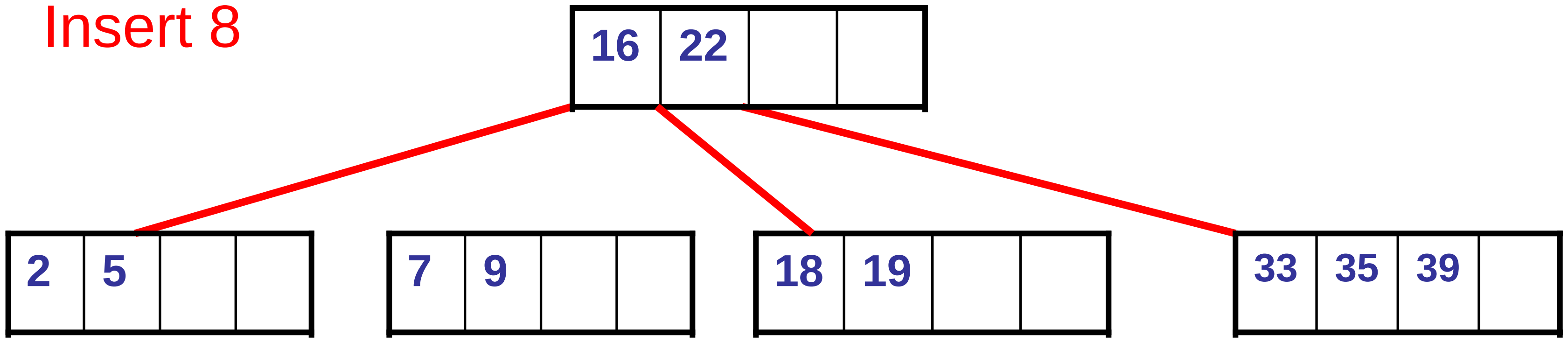


Split the leaf, creating a new leaf, and move half the keys from full leaf to new leaf.

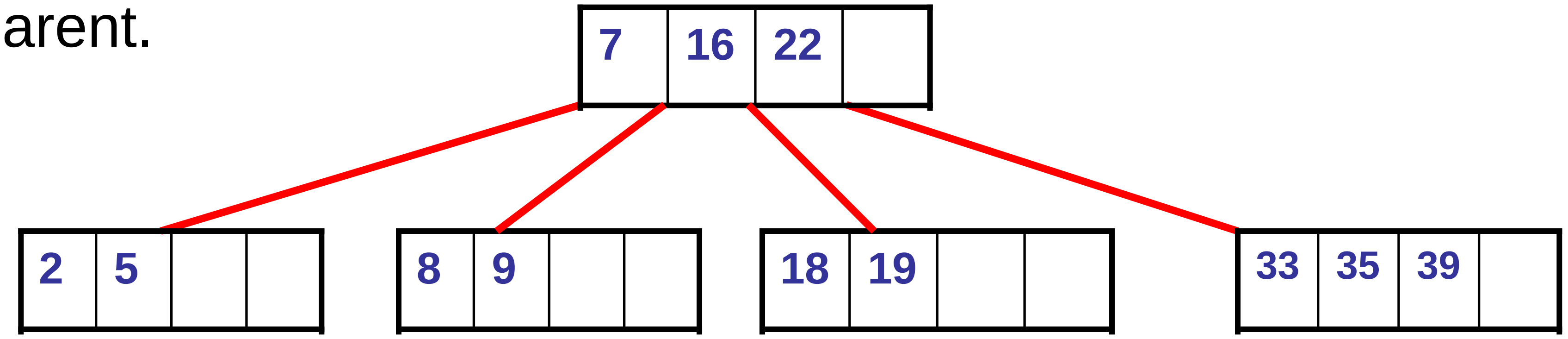


B-Tree Insertion: Case 2

Insert 8



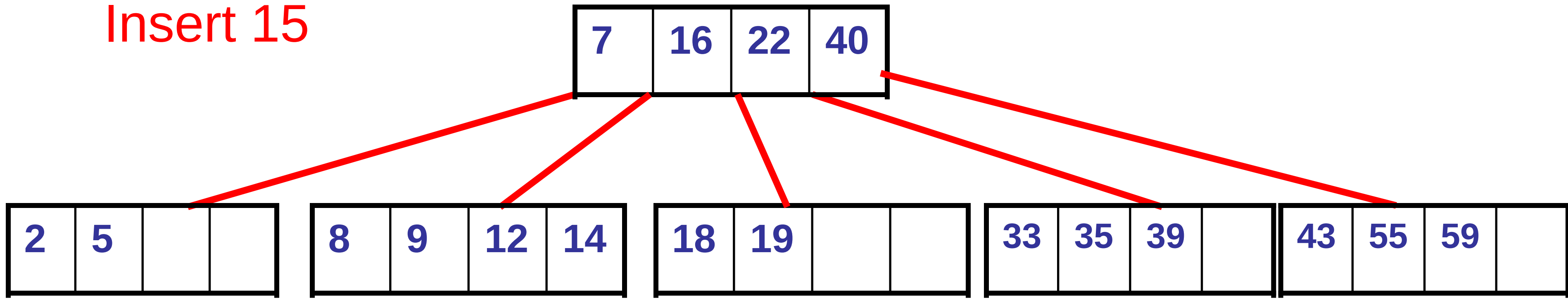
Move median key to parent, and add pointer to new leaf in parent.



B-Tree Insertion: Case 3

The root is full and must be split

Insert 15

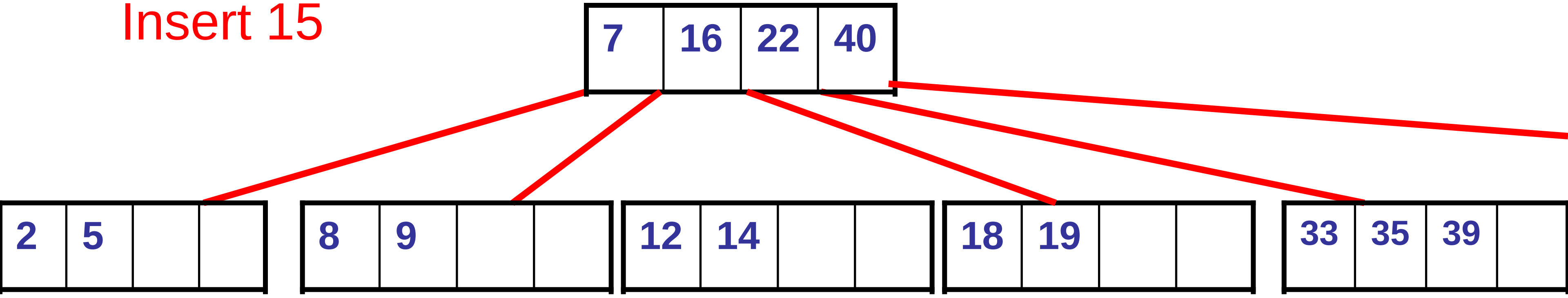


In this case, a new node must be created at each level, plus a new root. This split results in an increase in the height of the tree.

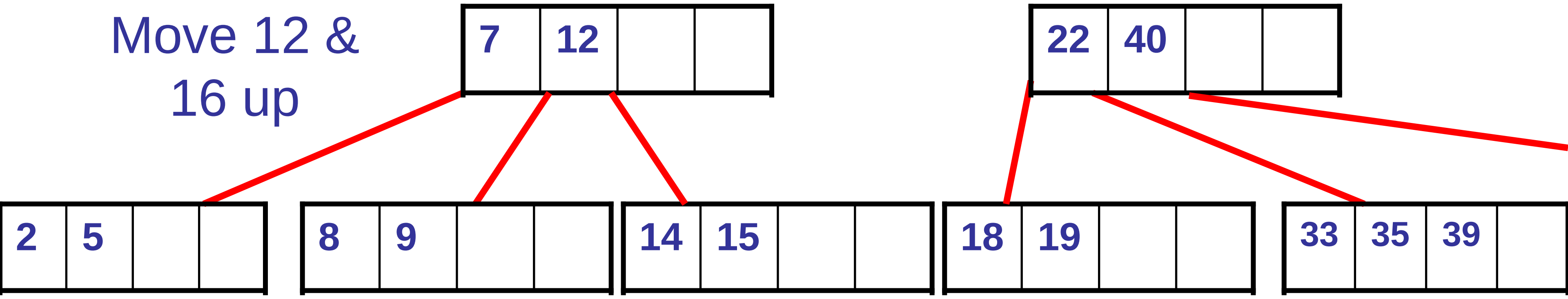
B-Tree Insertion: Case 3

The root is full and must be split

Insert 15

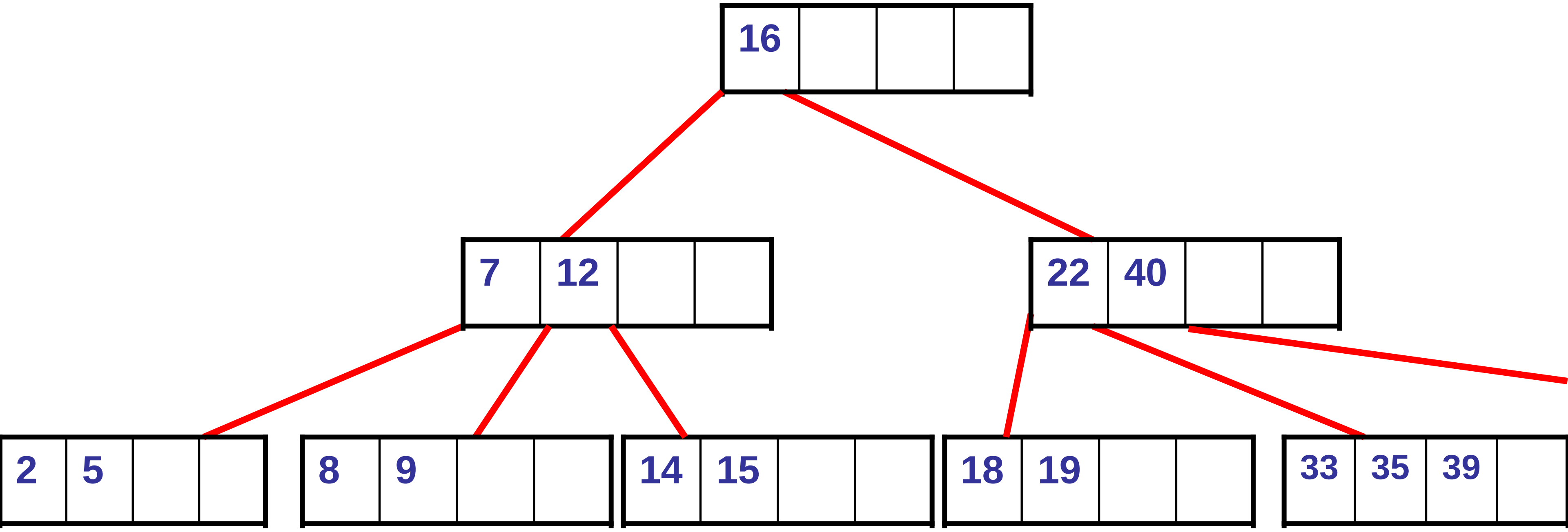


Move 12 & 16 up

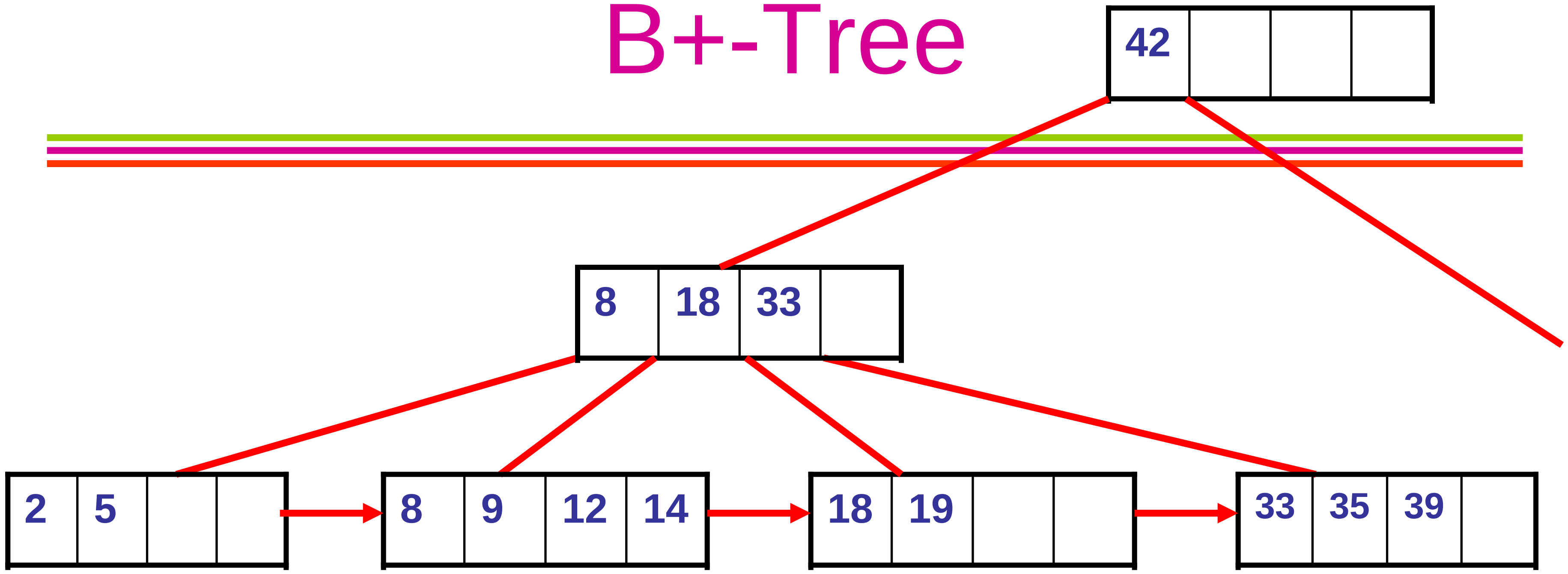


B-Tree Insertion: Case 3

This is the only case in which the height of the B-tree increases.



B+-Tree



A B+-Tree has all keys, with attached records, at the leaf level. Search keys, without attached records, are duplicated at upper levels. A B+-tree also has links between the leaves. Why?

B-Tree Insertion Analysis

Let M = Order of B-Tree

N = Number of Keys

B = Number of Keys that fit in one Block

Programmer defines size of node in tree to be ~1 block.

How many disk accesses to search for a key in the worst case?

Application: Web Search Engine

A web crawler program gathers information about web pages and stores it in a database for later retrieval by keyword by a search engine such as Google.

- Search Engine Task: Given a keyword, return the list of web pages containing the keyword.
- Assumptions:
  - The list of keywords can fit in internal memory, but the list of webpages (urls) for each keyword (potentially millions) cannot.
  - Query could be for single or multiple keywords, in which pages contain all of the keywords, but pages are not ranked.

What data structures should be used?