B-Trees, Slow Sorts

Data Structures and Parallelism

Announcements

Midterm Grades on gradescope

- Median 89/110
- Mean 90/110
- Standard Deviation: 11.07
- Solutions on webpage later today.
- Regrades will open on gradescope tomorrow.

Project 2 checkpoint Wednesday in lecture

Outline

Crash Course: Memory Hierarchy

B-Trees (our last dictionary)

Why we’re not implementing them

Start Sorting

Dictionaries Review

We've talked about two types of dictionaries:

AVL trees – $O(\log n)$ find, insert, and delete.

- Optimize for worst case

Hash Tables – $O(1)$ find, insert **on average, under assumptions**

- Optimize for average case.

One more design goal:

- Optimize behavior for huge datasets.
- Need to understand how memory works.

A False Assumption

We said “every operation takes about the same time”

In order to do big-O analysis.

Most of the time this is true.

It’s always true up to a constant factor.

- But what if that constant is 5,000,000?

Today: What to do when your dictionary is huge.

Memory Hierarchy

$$\begin{tikzcd}
\text{CPU Register} \arrow[d] \\
\text{L1 Cache} \arrow[d] \\
\text{L2 Cache} \arrow[d] \\
\text{RAM} \arrow[d] \\
\text{Disk}
\end{tikzcd}$$

What is it?	Typical Size	Time
Processor's memory	32 bits	≈free
Extremely fast	128KB	0.5 ns
Very fast	2MB	7 ns
what your programs need	8GB	100 ns
Large, longtime storage	1 TB	8,000,000 ns

Instructions vs. Memory Access

Let’s say the processor can do 2^{30} instructions per second.
How many instructions could you execute while waiting to hear back from memory?

Memory	Number of Instructions
L1 cache	2 instructions
L2 cache	30 instructions
Main memory	250 instructions
Disk	8,000,000 instructions.

Why have we not noticed this?

Very smart people work on algorithms to quickly decide what memory to put in caches (in case you need it again).

If your data fits in the cache, you'll probably never notice.

Compiler optimizations can sometimes ask for data before you need it.

When you ask for one piece of data, the OS will give you that, and everything near it.

- It's likely to be used soon (think arrays).

Once you use a value, the OS will keep it in a close by cache.

Two Principles

Temporal Locality:

- If you use some piece of memory, you are likely to use <u>that exact data</u> again pretty soon.

Spatial Locality:

- If you use some piece of memory, you are likely to use <u>nearby</u> data pretty soon.

OS accomplishes this by:

Keeping recently used memory in the cache

Moving memory in “pages” – if you access one data point, you move everything nearby with it.

Pause to Ponder

We’ve seen two dictionary types:

Search tree-based dictionaries

Table-based dictionaries

Which one should we use when we have a huge dataset?

Memory Accesses – Dictionaries

Suppose we have a dictionary with about 2^{50} elements.

Could store as an AVL tree of height about 50.

How many disk accesses might it take to `find`?

What if we made the tree shorter?

Make it an M-ary tree, not a binary tree. (How do we search?)

Disk accesses? Only $50 \log_M 2$ If $M = 30$, we'll cut the memory accesses to about 10. M will often be even bigger.

B-Trees

Reduce memory accesses:

Don’t put the values in a node, just the keys.

Make each node take up exactly one memory block.

We now have room for a bunch of keys, have a bunch of children.

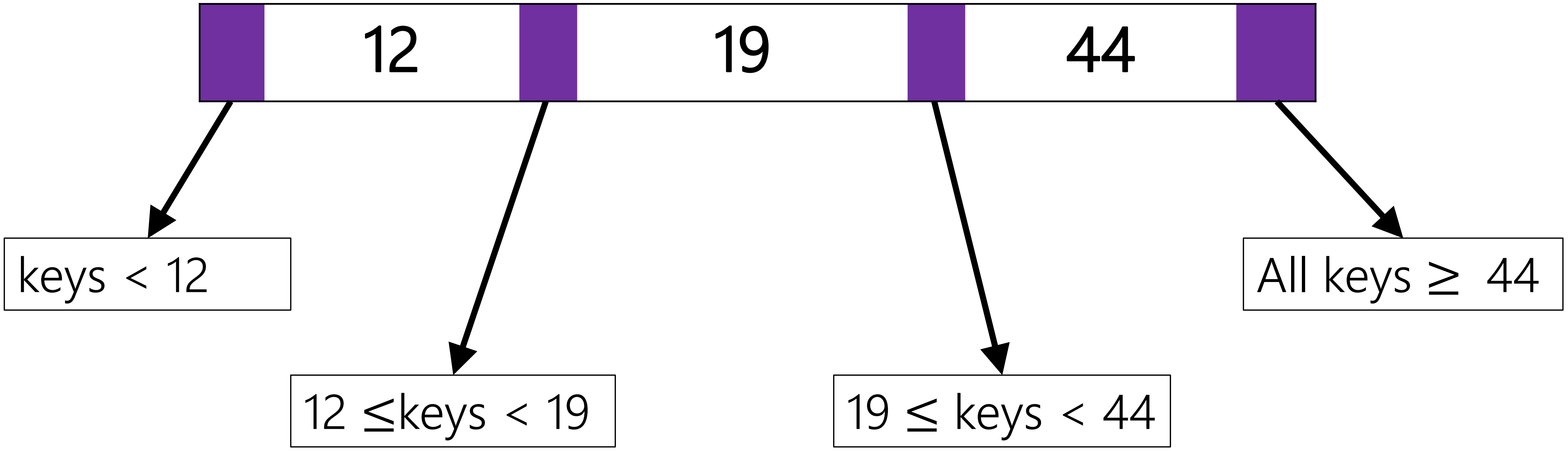
- Will make the tree shorter.

Put all the values in leaves.

- Again make each leaf take up exactly one memory block.

B-Trees

How do we search?



Binary search on “signpost” keys.
Entire node is in cache – binary search doesn’t need to go out to memory.

How Tall?

If the tree is “balanced” then we’ll get height about $\log_M(n)$

To maintain balance:

1. Root (special case):
   - Starts as a leaf (if almost empty – this should be very rare)
   - Otherwise has between 2 and M children
2. Every other internal node has between $[M/2]$ and M children.
3. Leaves have between $[L/2]$ and L (key, value) pairs.
4. Keep all leaves at same level.

Choose L and M as large as possible while still fitting in a block

Analysis

Memory accesses to `find, insert, delete`: about $\mathbf{log}_M(n)$

Much better than AVL trees as M gets large.

Running times? Need to think about the algorithms:

Insert Algorithm

`Find` then add to leaf.

If leaf is already full, then split leaf in two

- (still meet “at least half full condition)

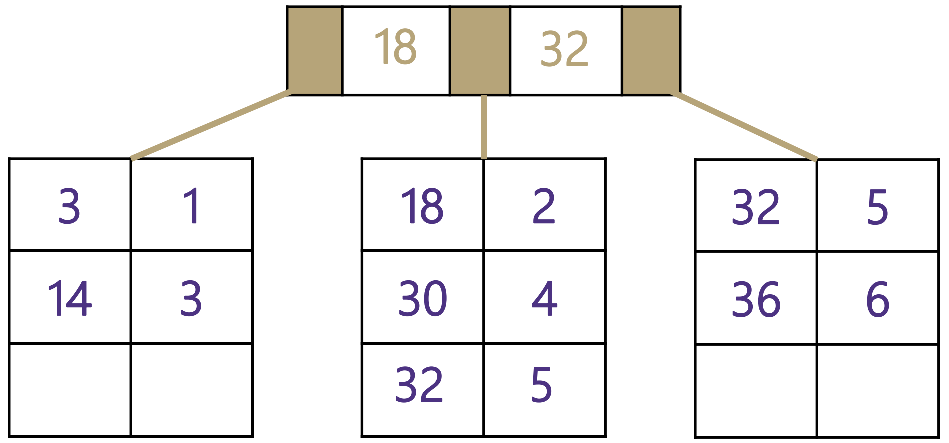
That adds a child to the parent node. What if it’s full?

Split that node in two!

Recurse up the tree (splitting as necessary)

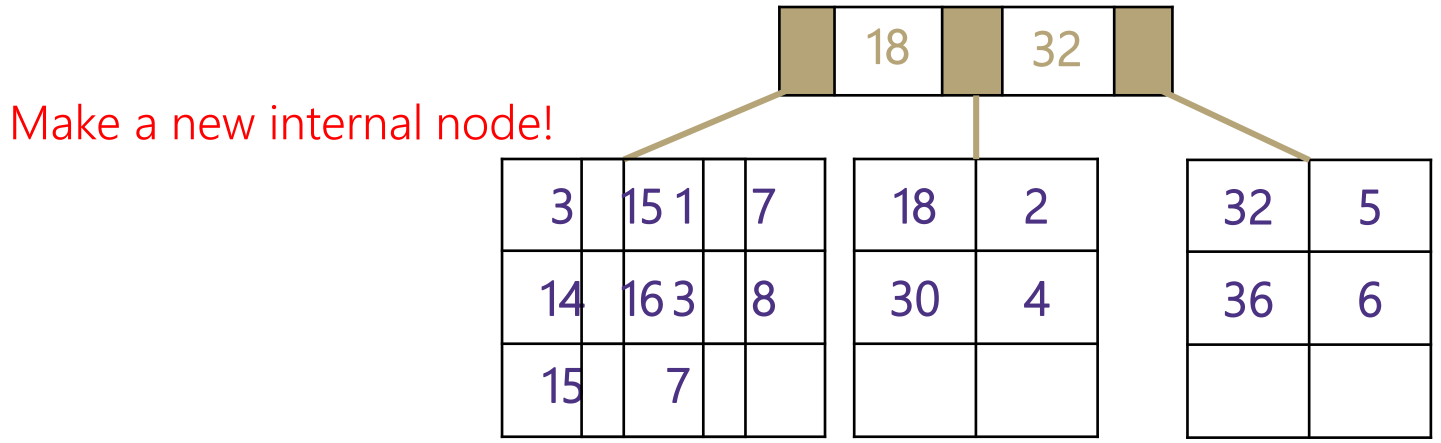
Insertion (simple)

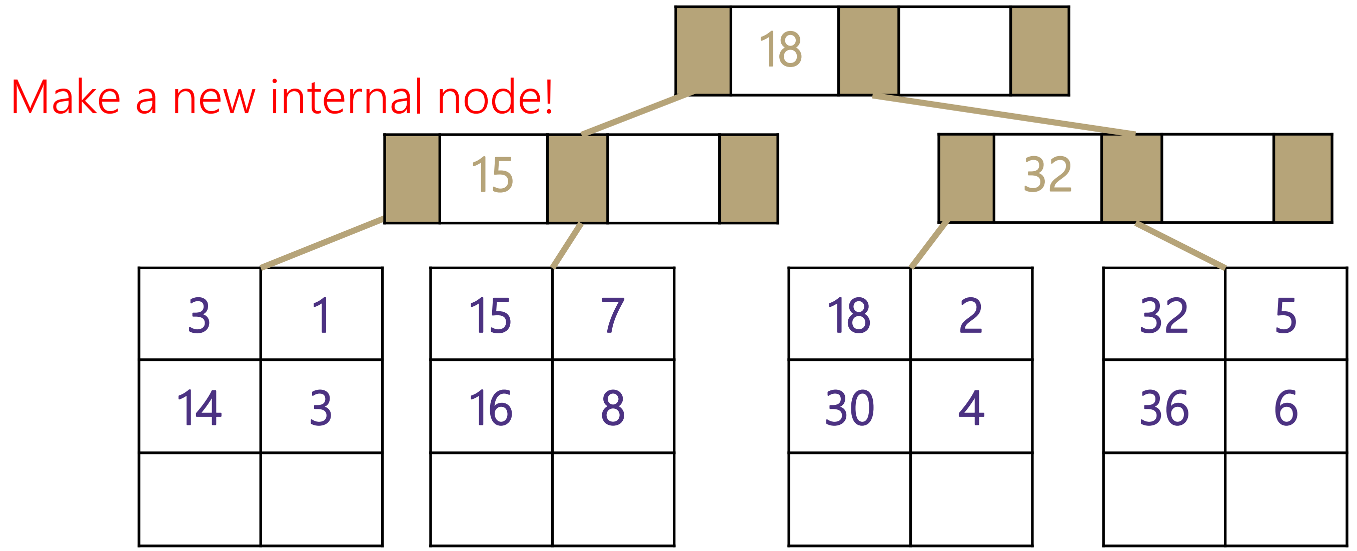
Try inserting (32, 5) and (36, 6) into the following tree



Insertion – splitting internal nodes

Try inserting (15, 7) and (16, 8) into our existing tree





Delete Algorithm

Find and remove from the relevant leaf.

If leaf "underflowed" to less than $\lceil L/2 \rceil$ elements:

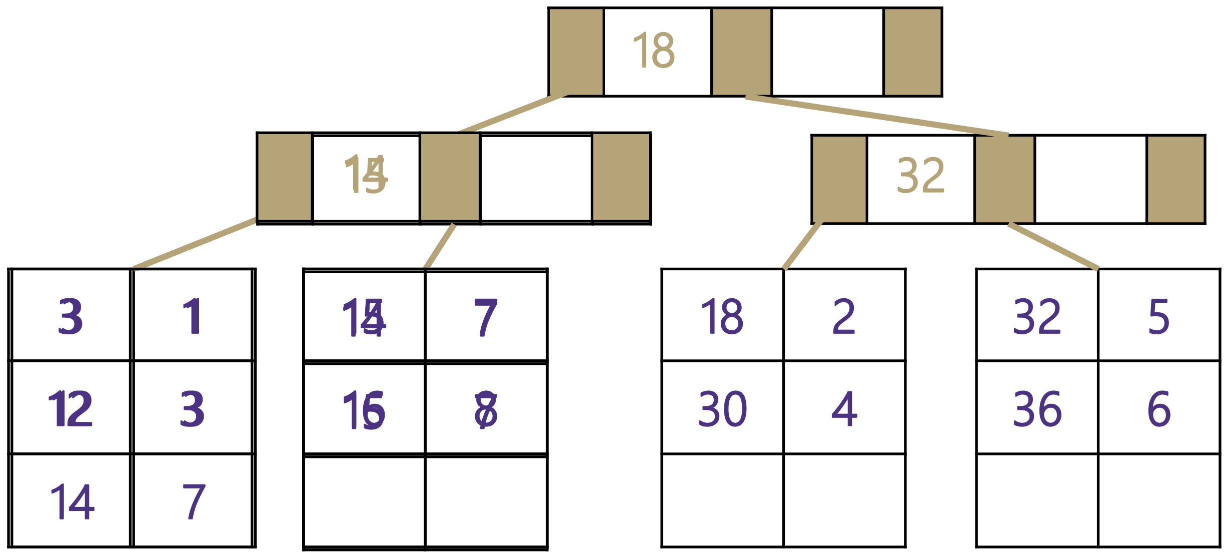
Try to "adopt" from a neighbor (then update signposts)

If neighbor is at exactly $\lceil L/2 \rceil$ elements, can't adopt. Merge instead.
- Have to recurse up the tree if we caused our parent to underflow.

If we cause the root to underflow to one child, delete it.
- This is the only way to make the height of the tree decrease.

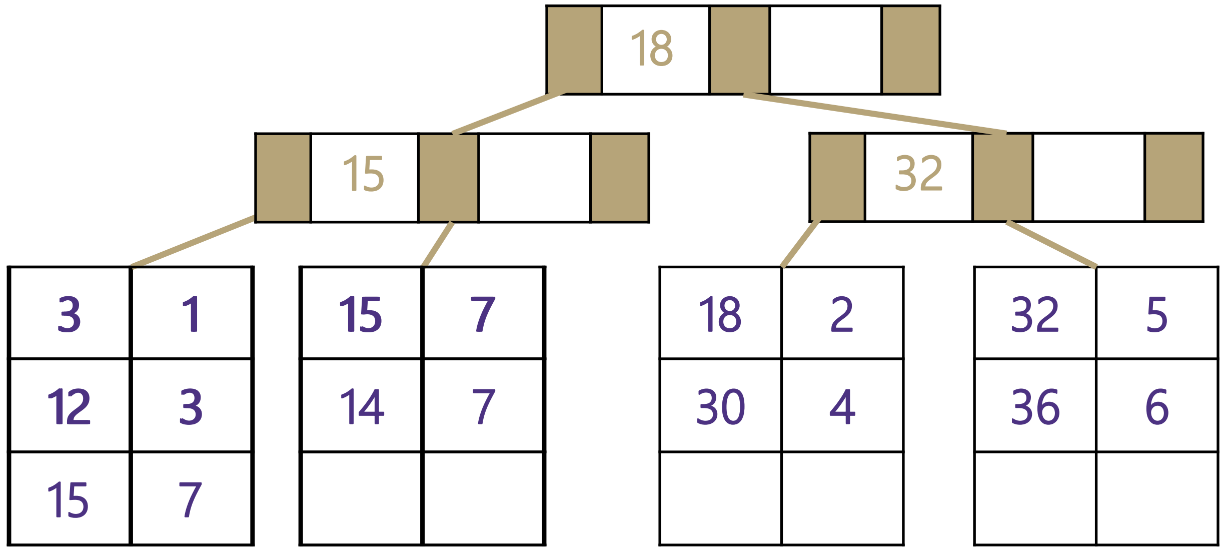
Deletion (simple)

Delete 16. Adopt from neighbor.
Keep leaf array sorted.



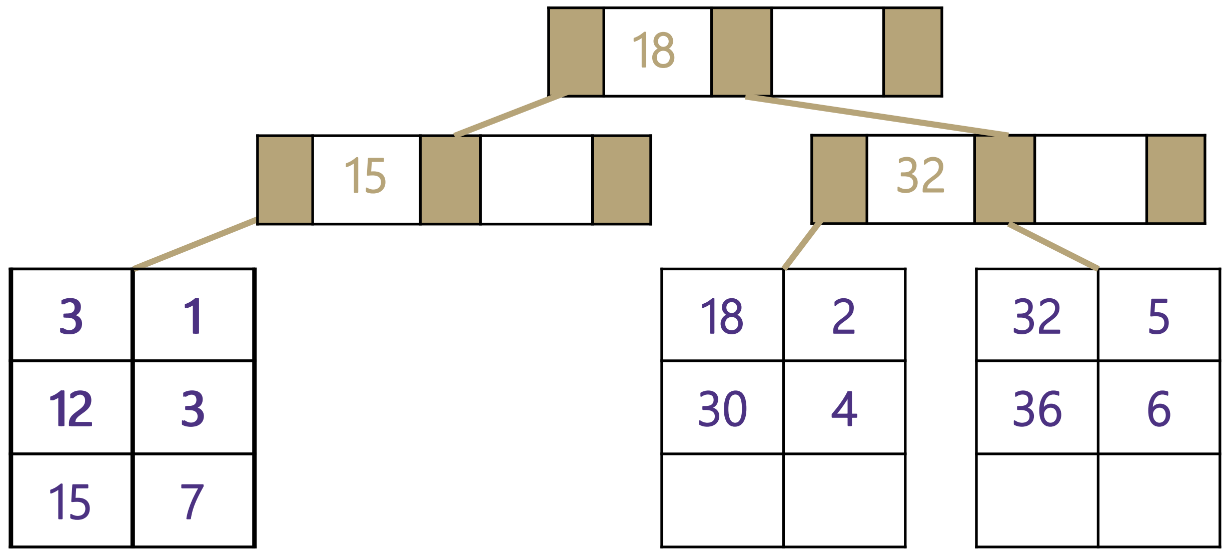
Deletion – merging nodes

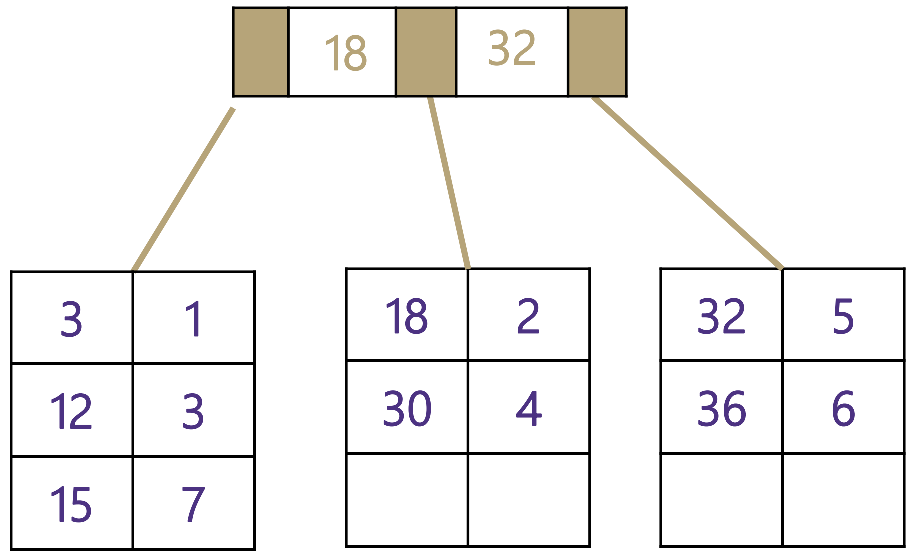
Delete 14. Can't adopt, so merge.



Deletion – merging nodes

Delete 14. Merge up the tree.
Update "signpost" to be smallest key in its right subtree.





Running times:

Insert (worst case):

Find correct leaf $O(\log_2 M \log_M n)$

insert in leaf $O(L)$

split leaf $O(L)$

split parents all the way to root $O(M \log_M n)$

Delete

Find correct leaf $O(\log_2 M \log_M n)$

Remove from leaf $O(L)$

Adopt/merge with neighbor $O(L)$

Adopt/merge all the way to root:$O(M \log_M n)$

Totals for both:
$O(M \log_M n + L)$

Remember, the point is not speed, but memory accesses!

B-Trees in Java?

If you want to implement a B-Tree, should you use Java?

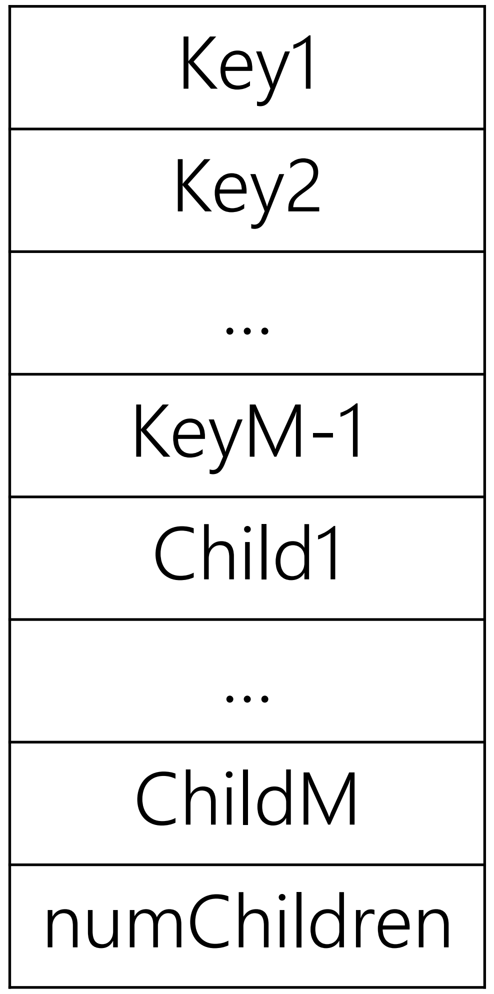
Java is designed to obscure details of the system you’re on from you.

But B-Trees need to know sizes of the memory pages.

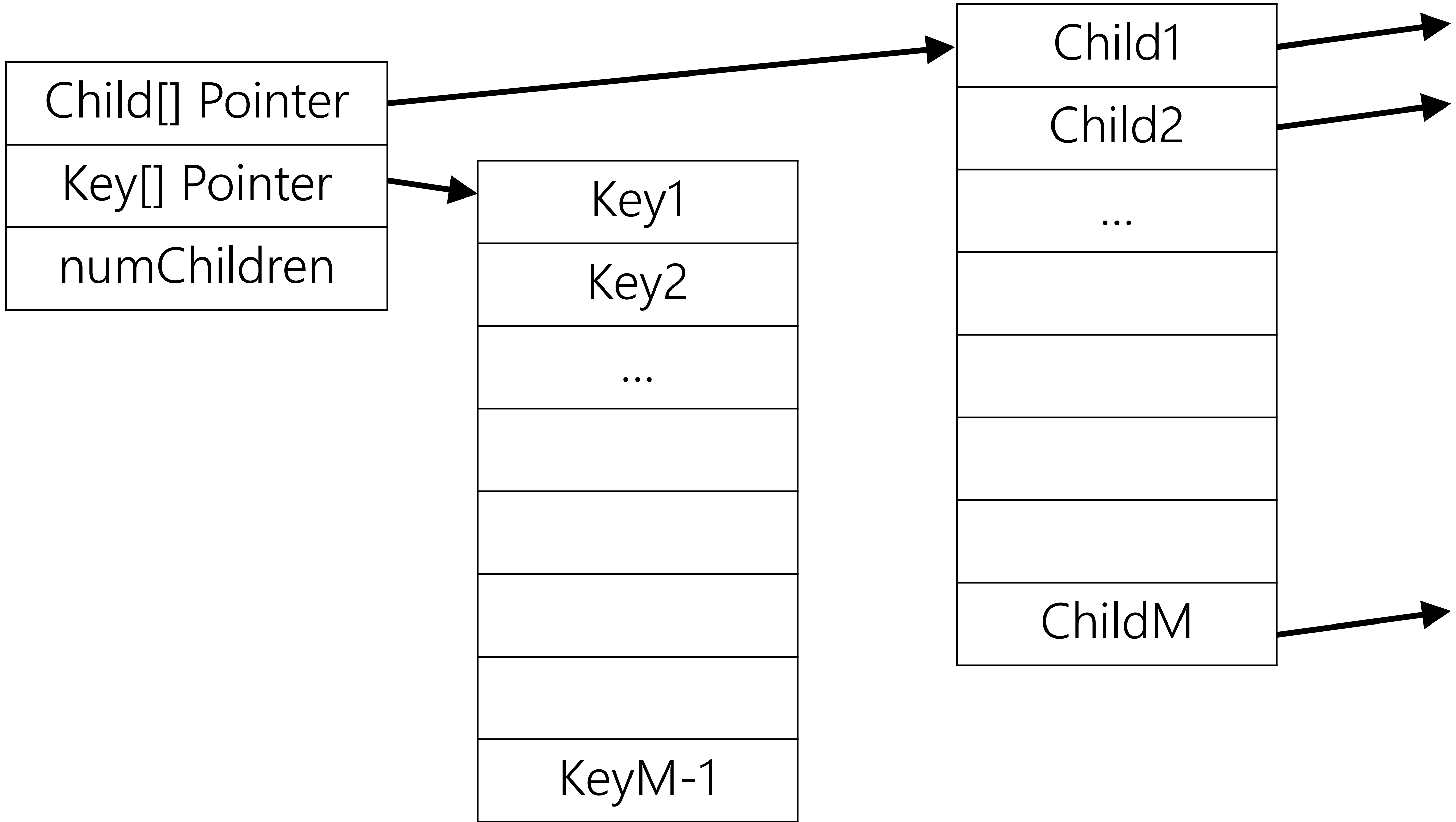
Picture of Java object

```
Class TreeNode{
      K[] keys
      TreeNode[] children
      int numChildren
      …
}
```

This is what we need our object to look like. Is this what it looks like?



What it Really Looks Like



Don’t Use Java

Even worse – Java uses “indirection” everywhere with objects.

- We wanted to stuff all the values in a single page.
- Not going to happen with Java Objects – everything is pointers

The JVM allocates the memory. No way to tell it to put things next to each other

If you want to implement a B-Tree.

Use C. (or C++ or some other lower-level language)

Sorting

Sorting

General Pre-processing Step

Let’s us find the k^{th} element in $O(1)$ time for any k.

Also a convenient way to discuss algorithm design principles.

Three goals

Three things you might want in a sorting algorithm:

In-Place

- Only use $O(1)$ extra memory.
- Sorted array given back in the input array.

Stable

- If a appears before b in the initial array and a.compareTo(b) == 0
- Then a appears before b in the final array.
- Example: sort by first name, then by last name.

Fast

Insertion Sort

How you sort a hand of cards.

Maintain a sorted subarray at the front.

Start with one element.

While(your subarray is not the full array)

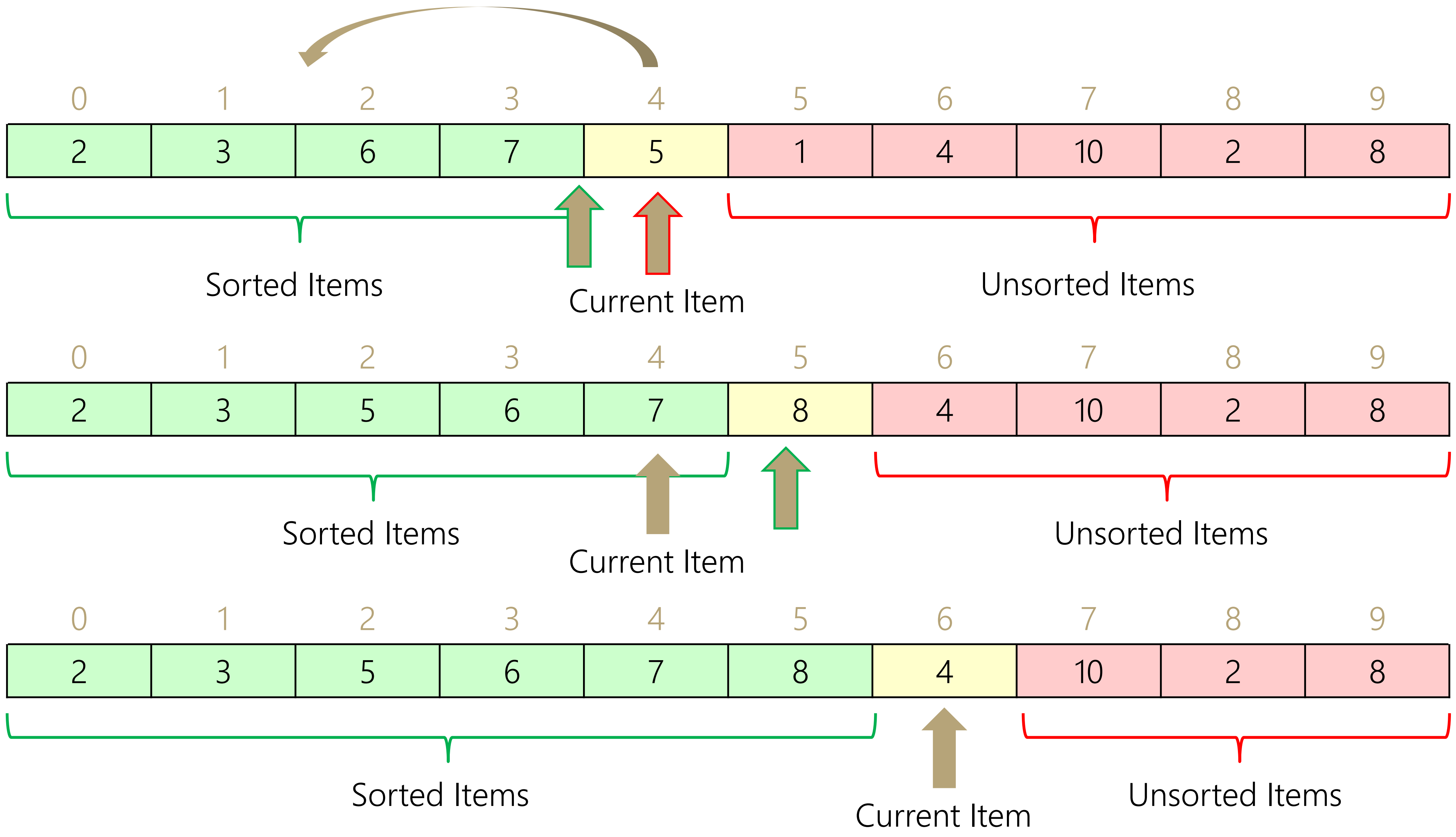
- Take the next element not in your subarray
- Insert it into the sorted subarray

Insertion Sort

```
for(i from 1 to n-1){
    int index = i
    while(a[index-1] > a[index]){
        swap(a[index-1], a[index])
        index = index-1
    }
}
```

Insertion Sort

https://www.youtube.com/watch?v=ROalU379l3U



Insertion Sort Analysis

Stable? Yes! (If you’re careful)

In Place Yes!

Running time:

- Best Case: $O(n)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$

Wednesday: Sorts with better Worst case behavior.