CSE 373

OCTOBER 23RD – MEMORY AND HARDWARE

MEMORY ANALYSIS

- Similar to runtime analysis

MEMORY ANALYSIS

- **Similar to runtime analysis**
  - Consider the worst case

MEMORY ANALYSIS

- **Similar to runtime analysis**
  - Rather than counting the number of operations, we count the amount of memory needed

MEMORY ANALYSIS

- **Similar to runtime analysis**
  - Rather than counting the number of operations, we count the amount of memory needed
  - During the operation, when does the algorithm need to "keep track" of the most number of things?

MEMORY ANALYSIS

- **Breadth first search**

MEMORY ANALYSIS

- **Breadth first search**
  - The Queue keeps track of the elements that need to be analyzed next.

MEMORY ANALYSIS

- **Breadth first search**
  - The Queue keeps track of the elements that need to be analyzed next.
  - This is the memory we need to consider

MEMORY ANALYSIS

- **Breadth first search**
  - The Queue keeps track of the elements that need to be analyzed next.
  - This is the memory we need to consider
  - At what point does the Queue have the *most* amount stored in it?

MEMORY ANALYSIS

- **Breadth first search**
  - The Queue keeps track of the elements that need to be analyzed next.
  - This is the memory we need to consider
  - At what point does the Queue have the *most* amount stored in it?
  - When the tree is at its widest – how many nodes is that?

MEMORY ANALYSIS

- **Breadth first search**
  - The Queue keeps track of the elements that need to be analyzed next.
  - This is the memory we need to consider
  - At what point does the Queue have the *most* amount stored in it?
  - When the tree is at its widest – how many nodes is that?
  - **N/2:** half the nodes of a tree are leaves

MEMORY ANALYSIS

- Consider finding an element in a sorted linked list

MEMORY ANALYSIS

- **Consider finding an element in a sorted linked list**
  - How much memory does this take?

MEMORY ANALYSIS

- **Consider finding an element in a sorted linked list**
  - How much memory does this take?
  - Don’t count the data structure, only count the amount of memory that the actual algorithm uses.

MEMORY ANALYSIS

- **Consider finding an element in a sorted linked list**
  - How much memory does this take?
  - Don’t count the data structure, only count the amount of memory that the actual algorithm uses.
  - What does it need to “keep track” of?

MEMORY ANALYSIS

- **Consider finding an element in a sorted linked list**
  - How much memory does this take?
  - Don’t count the data structure, only count the amount of memory that the actual algorithm uses.
  - What does it need to “keep track” of?
  - *Just the thing we’re looking for!*

MEMORY ANALYSIS

- **Consider finding an element in a sorted linked list**
  - How much memory does this take?
  - Don’t count the data structure, only count the amount of memory that the actual algorithm uses.
  - What does it need to “keep track” of?
  - *Just the thing we’re looking for!* O(1)

MEMORY ANALYSIS

- **We care about the asymptotic memory usage**

MEMORY ANALYSIS

- **We care about the asymptotic memory usage**
- **That is, as the input size of the data structures increases, does the amount of extra memory increase?**

MEMORY ANALYSIS

- **We care about the asymptotic memory usage**
- **That is, as the input size of the data structures increases, does the amount of extra memory increase?**
  - **AVL Insert?** Yes, we need to keep track of the path from the insertion to the root

HARDWARE CONSTRAINTS

- So far, we've taken for granted that memory access in the computer is constant and easily accessible

HARDWARE CONSTRAINTS

- **So far, we've taken for granted that memory access in the computer is constant and easily accessible**
  - This isn't always true!

HARDWARE CONSTRAINTS

- **So far, we've taken for granted that memory access in the computer is constant and easily accessible**
  - This isn't always true!
  - At any given time, some memory might be cheaper and easier to access than others

HARDWARE CONSTRAINTS

- **So far, we've taken for granted that memory access in the computer is constant and easily accessible**
  - This isn't always true!
  - At any given time, some memory might be cheaper and easier to access than others
  - Memory can't always be accessed easily

HARDWARE CONSTRAINTS

- **So far, we've taken for granted that memory access in the computer is constant and easily accessible**
  - This isn't always true!
  - At any given time, some memory might be cheaper and easier to access than others
  - Memory can't always be accessed easily
  - Sometimes the OS lies, and says an object is "in memory" when it's actually on the disk

HARDWARE CONSTRAINTS

- **Back on 32-bit machines, each program had access to 4GB of memory**

HARDWARE CONSTRAINTS

- **Back on 32-bit machines, each program had access to 4GB of memory**
    - This isn’t feasible to provide!

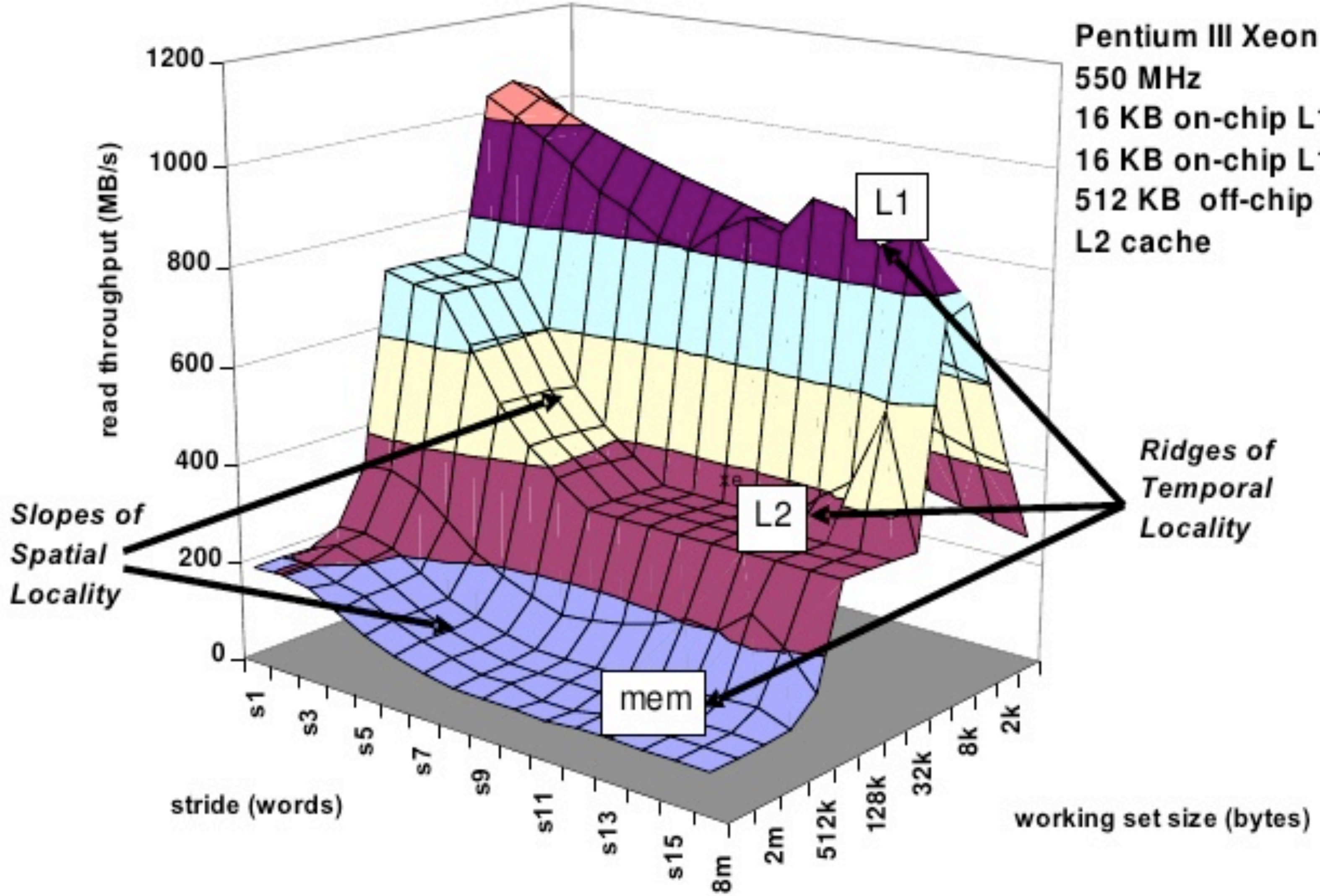
HARDWARE CONSTRAINTS

- **Back on 32-bit machines, each program had access to 4GB of memory**
  - This isn’t feasible to provide!
  - Sometimes there isn’t enough available, and so memory that hasn’t been used in a while gets pushed to the disk

HARDWARE CONSTRAINTS

- **Back on 32-bit machines, each program had access to 4GB of memory**
  - This isn’t feasible to provide!
  - Sometimes there isn’t enough available, and so memory that hasn’t been used in a while gets pushed to the disk
- **Memory that is frequently accessed goes to the cache, which is even faster than RAM**

The Memory Mountain



Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

LOCALITY AND PAGES

- **So, the OS does two smart things**
  - Spatial locality – if you use memory index Ox371347AB, you are likely to need Ox371347AC – bring both into cache
  - These are called pages, and they are usually around 4kb

LOCALITY AND PAGES

- **So, the OS does two smart things**
  - Spatial locality – if you use memory index Ox371347AB, you are likely to need Ox371347AC – bring both into cache
  - These are called pages, and they are usually around 4kb
  - All of the processes on your computer have access to pages in memory.

LOCALITY AND PAGES

- When you call new in Java, you are requesting new memory from the heap. If there isn't memory there, the JVM needs to get new memory from the OS

LOCALITY AND PAGES

- **When you call new in Java, you are requesting new memory from the heap. If there isn't memory there, the JVM needs to get new memory from the OS**
  - The OS only uses memory in page sizes

LOCALITY AND PAGES

- **When you call new in Java, you are requesting new memory from the heap. If there isn't memory there, the JVM needs to get new memory from the OS**
  - The OS only uses memory in page sizes
  - So if you allocate 100Bytes of data, you overallocate to 4kb!

LOCALITY AND PAGES

- **When you call new in Java, you are requesting new memory from the heap. If there isn't memory there, the JVM needs to get new memory from the OS**
  - The OS only uses memory in page sizes
  - So if you allocate 100Bytes of data, you overallocate to 4kb!
  - But you can use that 4kb if you need more

LOCALITY AND PAGES

- **Secondly, the OS uses temporal locality,**

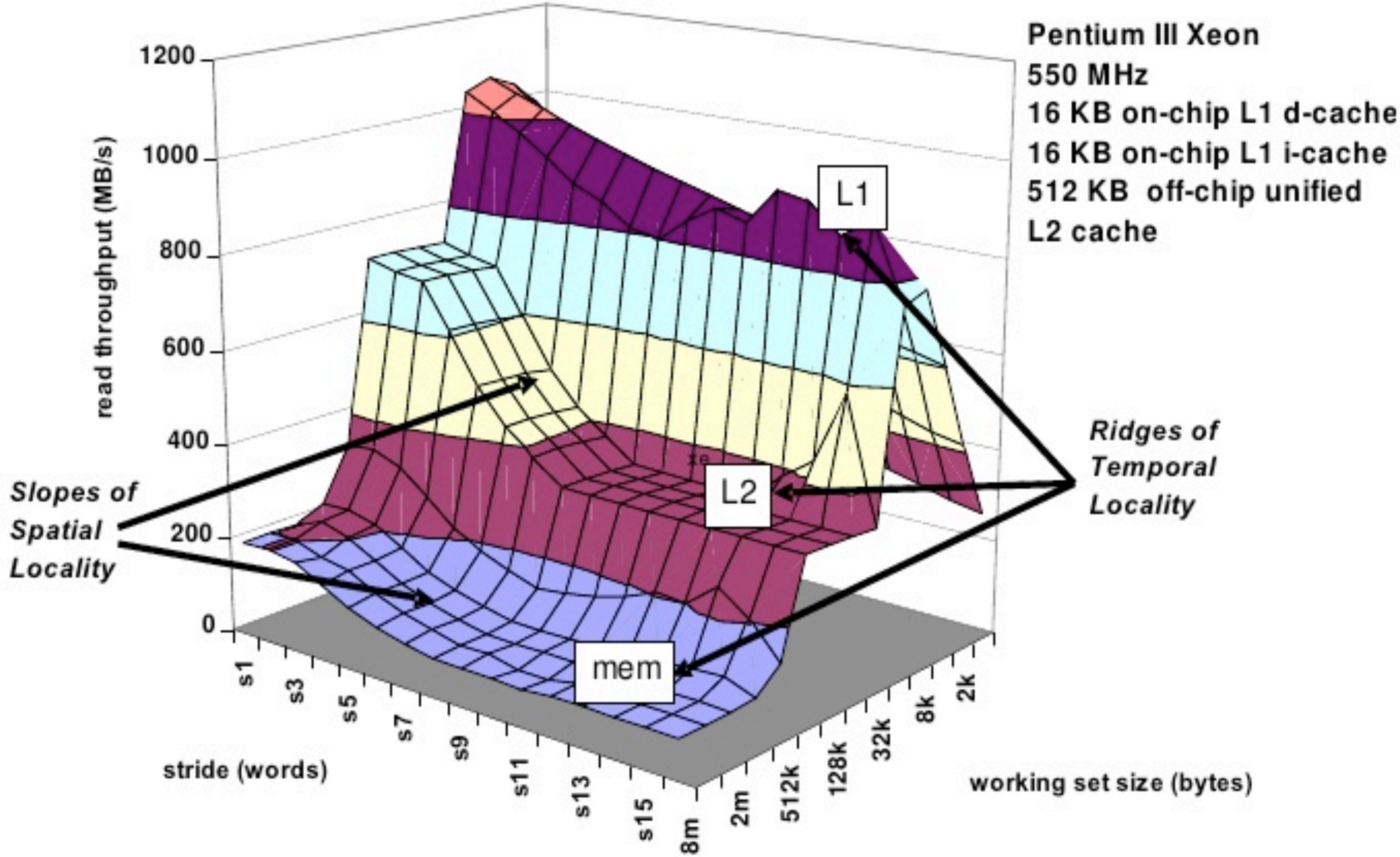
LOCALITY AND PAGES

- **Secondly, the OS uses temporal locality,**
  - Memory recently accessed is likely to be accessed again

LOCALITY AND PAGES

- **Secondly, the OS uses temporal locality,**
  - Memory recently accessed is likely to be accessed again
  - Bring recently used data into faster memory

The Memory Mountain



LOCALITY AND PAGES

- **Secondly, the OS uses temporal locality,**
  - Memory recently accessed is likely to be accessed again
  - Bring recently used data into faster memory
- **Types of memory (by speed)**
  - Register
  - L1,L2,L3
  - Memory
  - Disk
  - The interwebs (the cloud)

LOCALITY AND PAGES

- **The OS is always processing this information and deciding which is the best**
  - This is why arrays are faster in practice, they are always next to each other in memory

LOCALITY AND PAGES

- **The OS is always processing this information and deciding which is the best**
  - This is why arrays are faster in practice, they are always next to each other in memory
  - Each new node in a tree may not even be in the same page in memory!!

LOCALITY AND PAGES

- **The OS is always processing this information and deciding which is the best**
  - This is why arrays are faster in practice, they are always next to each other in memory
  - Each new node in a tree may not even be in the same page in memory!!
- **Important to consider when designing and explaining design problems.**

COST OF MEMORY ACCESSES

- Registers (128B): Instantaneous access
- L2 Cache (128KB): 0.5 nanoseconds
- L3 Cache (2MB): 7 nanoseconds
- Main Memory (32 GB): 100 nanoseconds

COST OF MEMORY ACCESSES

- **Registers (128B): Instantaneous access**
- **L2 Cache (128KB): 0.5 nanoseconds**
- **L3 Cache (2MB): 7 nanoseconds**
- **Main Memory (32 GB): 100 nanoseconds**
- ***Disk (TBs): 8,000,000 nanoseconds***

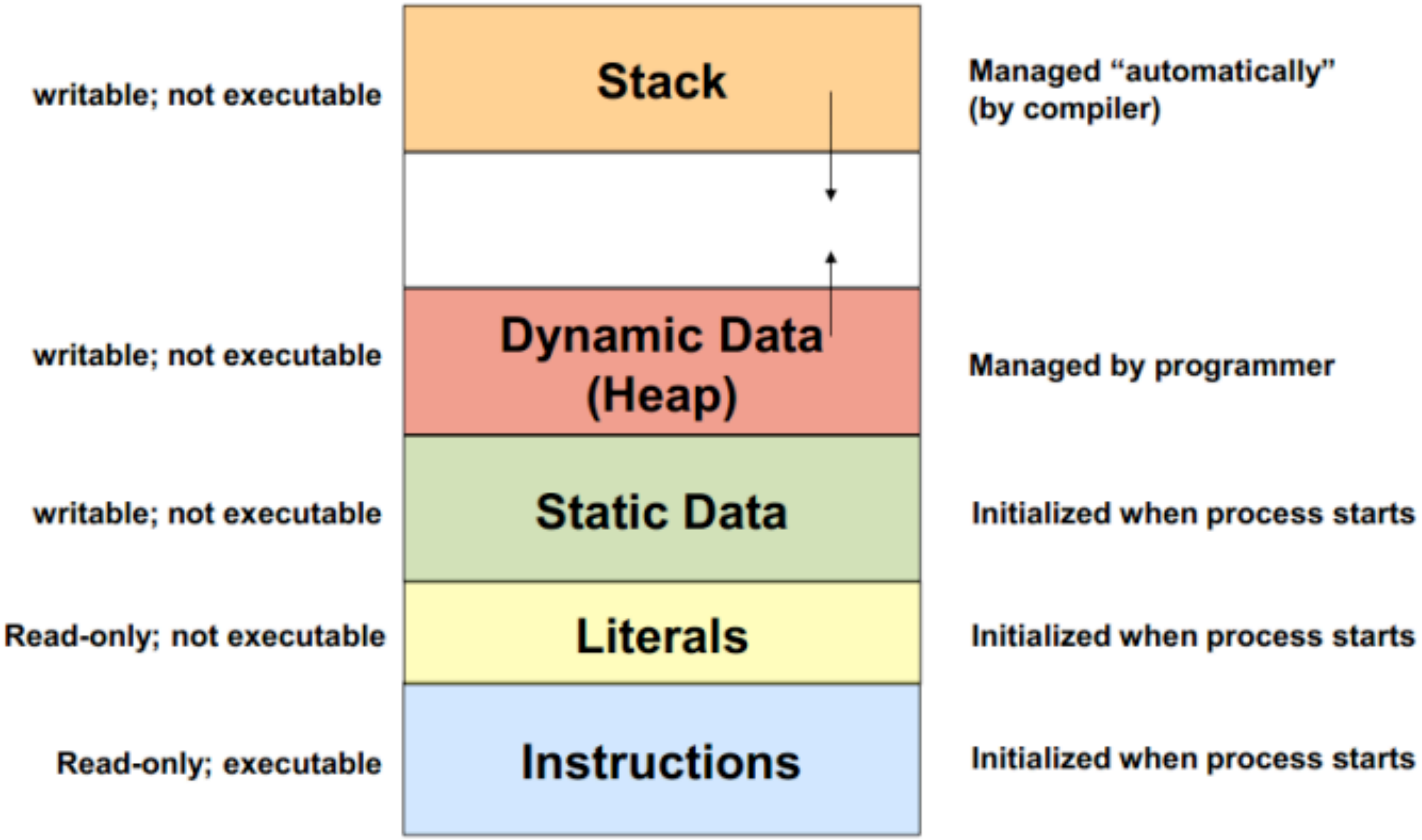
PROCESS MEMORY

- **How does an individual process use memory?**

PROCESS MEMORY

- **How does an individual process use memory?**
- **Many different demands**
  - Global variables
  - Call stack
  - Allocated variables
  - Process code

PROCESS MEMORY



PROCESS MEMORY

- These different demands are not next to each other in memory—little locality benefit

PROCESS MEMORY

- **These different demands are not next to each other in memory—little locality benefit**
- **Each call to new allocates wherever there is space in the heap (memory allocator)**
  - Even if two elements are created one after another, there is no guarantee that they'll both be in the same page
  - This is especially true for java
  - How important is caching?

COST OF MEMORY ACCESSES

- **Registers (128B): Instantaneous access**
- **L2 Cache (128KB): 0.5 nanoseconds**
- **L3 Cache (2MB): 7 nanoseconds**
- **Main Memory (32 GB): 100 nanoseconds**
- ***Disk (TBs): 8,000,000 nanoseconds***
  - *This is much, much worse*

LARGE AVL

- **Suppose we are storing terabytes of data in an AVL tree**

LARGE AVL

- **Suppose we are storing terabytes of data in an AVL tree**
  - Height is about 50

LARGE AVL

- **Suppose we are storing terabytes of data in an AVL tree**
  - Height is about 50
  - How many disk accesses will a find take?

LARGE AVL

- **Suppose we are storing terabytes of data in an AVL tree**
  - Height is about 50
  - How many disk accesses will a find take?
  - Between 0 and 50!

LARGE AVL

- **Suppose we are storing terabytes of data in an AVL tree**
  - Height is about 50
  - How many disk accesses will a find take?
  - Between 0 and 50!
  - This is the difference between nanoseconds and almost half a second!

LARGE AVL

- **Suppose we are storing terabytes of data in an AVL tree**
  - Height is about 50
  - How many disk accesses will a find take?
  - Between 0 and 50!
  - This is the difference between nanoseconds and almost half a second!
  - If lots data is stored on the disk, O(log n) finds don’t happen in practice

PROBLEMS

- **Why is AVL so bad on disk?**

PROBLEMS

- **Why is AVL so bad on disk?**
  - Each piece of data is its own node

PROBLEMS

- **Why is AVL so bad on disk?**
  - Each piece of data is its own node
  - Each call of `new` may not place objects next to each other

PROBLEMS

- **Why is AVL so bad on disk?**
  - Each piece of data is its own node
  - Each call of **`new`** may not place objects next to each other
  - Has large height, for the number of elements?

SOLUTIONS

- **What changes might we want to make to an AVL to make it better for disk?**

SOLUTIONS

- **What changes might we want to make to an AVL to make it better for disk?**
  - Still want to keep log n height

SOLUTIONS

- **What changes might we want to make to an AVL to make it better for disk?**
  - Still want to keep log n height
  - Allocate more objects closer together

SOLUTIONS

- **What changes might we want to make to an AVL to make it better for disk?**
  - Still want to keep log n height
  - Allocate more objects closer together
  - Have a higher branching factor so that data you want is at a lower depth

SOLUTIONS

- **What changes might we want to make to an AVL to make it better for disk?**
  - Still want to keep log n height
  - Allocate more objects closer together
  - Have a higher branching factor so that data you want is at a lower depth
  - Take advantage of page sizes