Announcements

- PS5 due 11/8 at 11:59PM
- Quiz #4 Thursday?
- Prelim #2 a week from today, review session probably on Saturday

- Anonymous survey comments
    - Problem sets about the same as usual, probably a bit shorter in fact

- Grade distribution (from lecture #1):

Breakdown:

- 40% -Problem sets (later ones count more)
- 5% -Quizzes (lowest dropped)
- 30% -Preliminary exams (lowest weighted less)
- 25% -Final exam

B-Trees and Cache-friendly data structures

- Last lecture: memory hierarchy and performance
- Basic lesson: locality is good (temporal as well as spatial), indirection is bad
    - This is in terms of performance, not correctness!

- How the compiler uses memory is never part of the formal spec, but basic structure is implicitly used so widely that it's essentially fixed
    - How basic types are represented, when indirections occur, are typically known by sophisticated users
    - Without this information it's impossible to write cache-friendly code
        - Example from SmallTalk

- In OCaml we can assume that tuples are in consecutive memory locations
    - Same as arrays, as discussed in last lecture
    - Note that we can still have indirection, if the elements are unboxed

- How does the cache play out with our favorite data structures, and how can we do better?
    - Note that, unusually for CS3110, this set of issues is easiest to understand in a "low-level" language like C, where the mapping to machine architecture is explicit
    - But even this is problematic as architecture changes and becomes more complex!

Lists

- Lists have a lot of pointers. Short lists will fit into cache, but long lists won't.
- So storing large sets in lists tends to result in a lot of cache misses.

Trees

- Trees have a lot of indirections, which can cause cache misses.
- On the other hand, the top few levels of the tree are likely to fit into the cache, and because they will be visited often, they will tend to be in the cache.
  - Cache misses will tend to happen at the less frequently visited nodes lower in the tree.
- One nice property of trees is that they preserve locality.
  - Keys that are close to each other will tend to share most of the path from the root to their respective nodes.
  - Further, an in-order tree traversal will access tree nodes with good locality.
- So trees are especially effective when they are used to store sets or maps where accesses to elements/keys exhibit good locality.

Hash tables

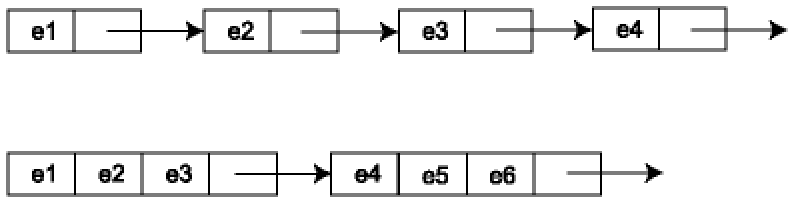
- Hash tables are arrays, but a good hash function destroys much locality by design. Accessing the same element repeatedly will bring the appropriate hash bucket into cache. But accesses to keys that are near each other in any ordering on keys will result in memory accesses with no locality.
  - If keys have a lot of locality, a tree data structure may be faster even though it is asymptotically slower!

Chunking

- Lists use the cache ineffectively.
  - Accessing a list node causes extra information on the cache line to be pulled into cache, possibly pushing out useful data as well.

- For best performance, cache lines should be used fully.

- Often data structures can be tweaked to make this happen.
- For example, with lists we might put multiple elements into each node.
- The representation of a bucket set is then a linked list where each node in the linked list contains several elements (and a chaining pointer) and takes up an entire cache line. Thus, we go from a linked list that looks like the one on top to the one on the bottom:



- Doing this kind of performance optimization can be tricky in a language like Ocaml where the language is working hard to hide these kind of low-level representation choices from you.
- A rule of thumb, however, is that OCaml records and tuples are stored nearby in memory.
- So this kind of memory layout can be implemented to some degree in OCaml.
    - The idea is that a chunk stores four elements and a counter that keeps track of the number of element fields (1-4) that are in use.
    - See code in lecture notes

```
type 'a onelist = Null | Cons of 'a * 'a onelist
type 'a chunklist = NullChunk
       | Chunk of 'a * 'a * 'a * 'a * int * 'a chunklist

let cons x (y: 'a chunklist) = match y with
      Chunk(_,_,_,a,1,tail) -> Chunk(x,x,x,a,2,tail)
    | Chunk(_,_,b,a,2,tail) -> Chunk(x,x,b,a,3,tail)
    | Chunk(_,c,b,a,3,tail) -> Chunk(x,c,b,a,4,tail)
    | (Chunk(_,_,_,_,_,_) | NullChunk) -> Chunk(x,x,x,x,1,y)

let rec find (l:int chunklist) (x:int) : bool =
    match l with
      NullChunk -> false
      | Chunk(a,b,c,d,n,t) ->
        (a=x || b=x || c=x || d=x || find t x)
```

- It turns out that scanning down a linked list implemented in this way is about 10 percent faster than scanning down a simple linked list (at least on a 2009 laptop).
- In a language like C or C++, programmers can control memory layout better and can reap much greater performance benefits by taking locality into account.
- The ability to carefully control how information is stored in memory is perhaps the best feature of these languages.

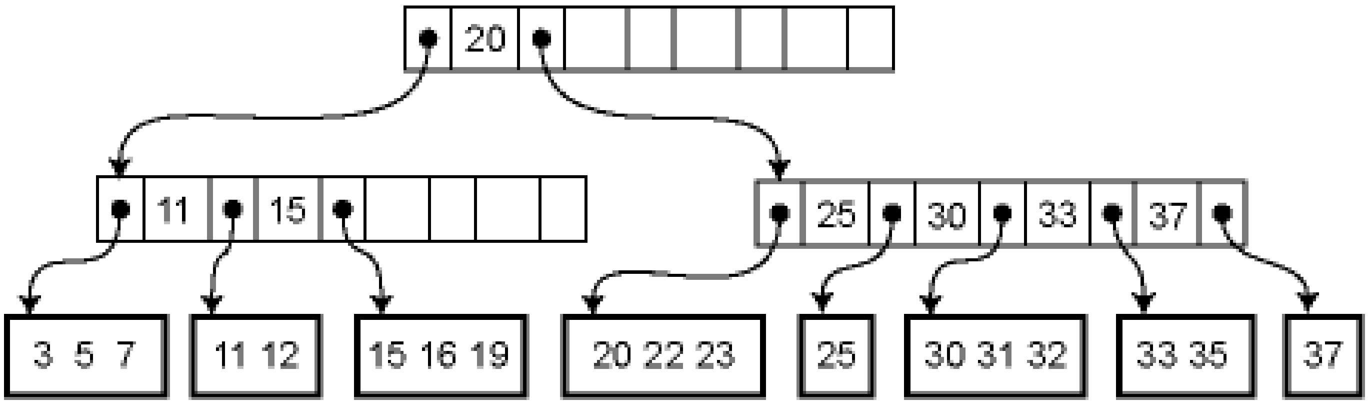
B trees

- The idea we saw earlier of putting multiple set (list, hash table) elements together into large chunks that exploit locality can also be applied to trees.
    - Binary search trees are not good for locality because a given node of the binary tree probably occupies only a fraction of any cache line.
    - **B-trees** are a way to get better locality by putting multiple elements into each tree node.

- B-trees were originally invented for storing data structures on disk, where locality is even more crucial than with memory.
    - Accessing a disk location takes about 5ms = 5,000,000ns.
    - Therefore, if you are storing a tree on disk, you want to make sure that a given disk read is as effective as possible.
- B-trees have a high branching factor, much larger than 2, which ensures that few disk reads are needed to navigate to the place where data is stored.
    - B-trees may also useful for in-memory data structures because these days main memory is almost as slow relative to the processor as disk drives were to main memory when B-trees were first introduced!

- A B-tree of order *m* is a search tree in which each nonleaf node has up to *m* children.
    - The actual elements of the collection are stored in the leaves of the tree, and the nonleaf nodes contain only keys.
- Each leaf stores some number of elements; the maximum number may be greater or (typically) less than *m*. The data structure satisfies several invariants:
    1. Every path from the root to a leaf has the same length
    2. If a node has *n* children, it contains *n*−1 keys.
    3. Every node (except the root) is at least half full
    4. The elements stored in a given subtree all have keys that are between the keys in the parent node on either side of the subtree pointer. (This generalizes the BST invariant.)
    5. The root has at least two children if it is not a leaf.

- Here is an order-5 B-tree (*m*=5) where the leaves have enough space to store up to 3 data records:

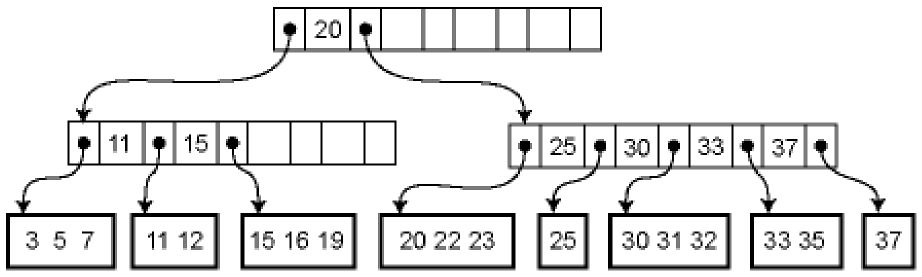


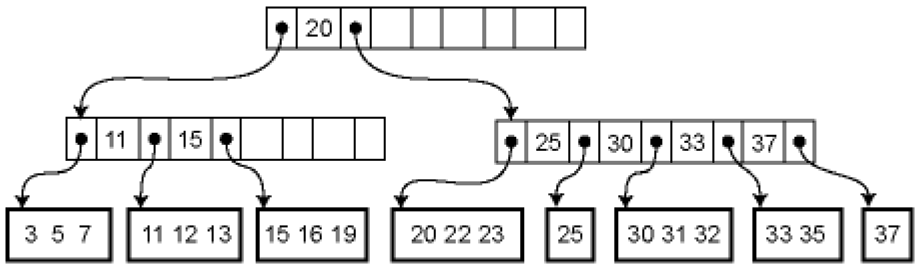
- Because the height of the tree is uniformly the same and every node is at least half full, we are guaranteed that the asymptotic performance is O(lg *n*) where *n* is the size of the collection.

- The real win is in the constant factors, of course. We can choose *m* so that the pointers to the *m* children plus the *m*−1 elements fill out a cache line at the highest level of the memory hierarchy where we can expect to get cache hits.
    - For example, if we are accessing a large disk database then our "cache lines" are memory blocks of the size that is read from disk.

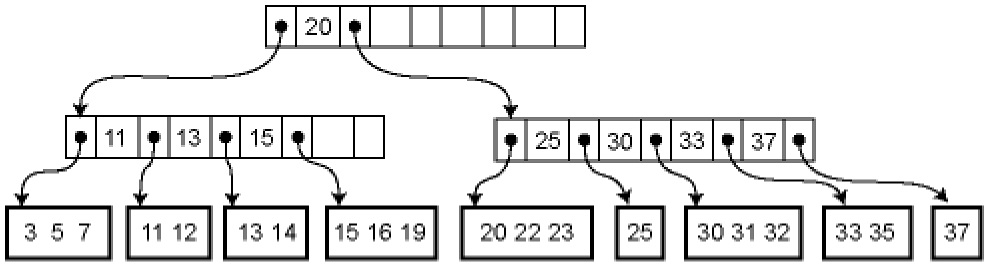
- Lookup in a B-tree is straightforward.
    - Given a node to start from, we use a simple linear or binary search to find whether the desired element is in the node, or if not, which child pointer to follow from the current node.

- Insertion and deletion from a B-tree are more complicated; in fact, they are notoriously difficult to implement correctly.
- For insertion, we first find the appropriate leaf node into which the inserted element falls (assuming it is not already in the tree).
    - If there is already room in the node, the new element can be inserted simply.
    - Otherwise the current leaf is already full and must be split into two leaves, one of which acquires the new element.
- The parent is then updated to contain a new key and child pointer. If the parent is already full, the process ripples upwards, eventually possibly reaching the root.
    - If the root is split into two, then a new root is created with just two children, increasing the height of the tree by one.
- For example, here is the effect of a series of insertions. The first insertion (13) merely affects a leaf. The second insertion (14) overflows the leaf and adds a key to an internal node. The third insertion propagates all the way to the root.

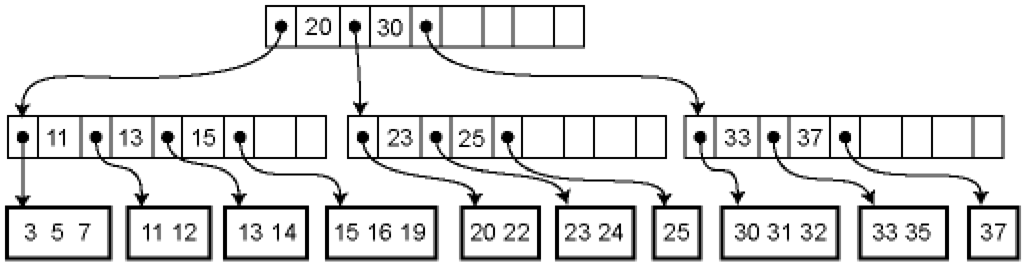




insert(13)



insert(13); insert(14)



insert(13); insert(14); insert(24)

- Deletion works in the opposite way: the element is removed from the leaf.
- If the leaf becomes empty, a key is removed from the parent node.
    - If that breaks invariant 3, the keys of the parent node and its immediate right (or left) sibling are reapportioned among them so that invariant 3 is satisfied.
    - If this is not possible, the parent node can be combined with that sibling, removing a key another level up in the tree and possible causing a ripple all the way to the root.
    - If the root has just two children, and they are combined, then the root is deleted and the new combined node becomes the root of the tree, reducing the height of the tree by one.

Further reading: Aho, Hopcroft, and Ullman, *Data Structures and Algorithms*, Chapter 11.