Trees, continued

- **Review of binary search**
  - Data is maintained in an array
  - Sorted to speed up search
  - Code

- **Symbol tables**
  - Dictionaries in Python, Maps in Java
  - Data values stored using keys
  - Examples: dictionary, phone directory, address book, etc.
    - Dictionary keys: words. Values: definitions
    - Phone directory keys: names. Values: numbers
    - Simple change from set implementations we’ve seen
    - Modify ArraySet code

- **Choices for organizing sorted collections of data (sets or maps)**
  1. Array
  2. Balanced binary tree
  3. B-tree (to be discussed below)

Which to choose, and why? -- the answer has to do in part with the way computer memory is organized

- **Computer Memory**
  - There are different kinds of computer memory
  - Trade-off between speed and cost
  - Therefore, memory is organized in a hierarchy
    - Cache memory (fast)
    - Main memory (~10 times slower)
    - Disk memory (~1000 times slower)
  - The faster the kind of memory, the less there is on a computer
- Computer memory organization
  - All data/programs are initially on the disk
  - When a program starts to execute, a portion of it (perhaps all of it) is copied into main memory
As the program runs, smaller portions are copied into the cache.

- Finally, data in the cache can be used by the computer's main processor.
- If the right data/code is in the cache, then most of the time the CPU can interact with fast memory, and access to slower memory is infrequent.
• **Arrays**
  - Arrays occupy contiguous locations in memory
  - Therefore, when a cell in an array is used, nearby cells will also be copied from disk into main memory and ultimately into the cache
  - Accessing cells in an array that are close together is relatively fast.
  - However, if the array is too big, and if accesses are not close together, this may result in page faults / cache misses, thereby slowing down the process

• **Trees**
  - The nodes in a tree do **not** necessarily occupy contiguous locations in memory
  - This is due to the way that "heap" memory is managed
    - Each time you use the `new` operator in Java, a portion of the heap is allocated as the memory in which object data is stored
    - If a tree is large enough, it can't all fit in main memory at once
    - Therefore, when a tree node is accessed, there is a higher probability of a page fault or cache miss
    - Moral: an application that uses a large dataset might be faster if an array is used, rather than a tree

• **Memory and data structure choice**
  - Accessing a node that is not in memory has great efficiency issues
    - Each disk access may take about 1000 times longer than accessing main memory, and copying from main memory to the cache may take as much as 10 times longer than interacting directly with the cache
  - Since nodes may not be stored in contiguous memory, it is best for large datasets that nodes be designed to take up a page

• **B-Trees**
  - Another implementation of sorted data
  - A generalization of 2-3 trees
  - Typically used to implement a symbol table
  - This time, all data items are stored in leaves
  - Leaves are "external" nodes (data may be stored in disk, or even on another computer in a network such as the Web)
  - Internal nodes consist of keys only
- **B-tree definition**
  - A B-tree of order $M$ (positive integer) is a tree in which each node has between $M/2$ and $M-1$ keys, and between $M/2+1$ and $M$ children (with the possible exception of the root)
    - If $M = 6$, then there are between 3 and 5 keys per node
    - Each key determines which child to search next
    - Example: text, p. 869

- **B-tree node**
  ```java
class Node<K,V> implements Comparable<K> {
    private static int M = 6; // or whatever value M is
    private K[] keys = new K[M];
    private V[] values;
    private Node<K,V>[] children = new Node<K,V>[M];
    private int size; // between M/2 and M-1
    private boolean isLeaf;
}
```

- **B-tree search**
  ```java
  public V get(K key) {
    return get(root, key); // root is a Node
  }

  public V get(Node<K,V> node, K key) {
    if (node.isLeaf)
      for (int i=0; i < node.size; i++)
        if (key.equals(keys[i]))
          return values[i];
    else {
      for (int i=0; i < node.size-1; i++) {
        int cmp = key.compareTo(node.keys[i]);
        if (cmp <= 0)
          return get(children[i], key);
      }
      if (key.compareTo(node.keys[size-1] <= 0))
        return get(node.children[size-2], key);
      else return get(node.children[size-1], key);
    }
  }
  }
```

- **Adding new data**
  - Find the leaf where the new data belongs
  - If the leaf is full (size $M$), split it
  - That may cause its parent to be full; split it
  - And so on up to the root
  - If the root is full, create a new root with 2 children (only exception to the requirement of at least $M-1$ children)
public V put(K key, V val) {
    return put(root, key, val);
}

public K put(Node<K,V> node, K key, V val) {
    if (node.isLeaf) {
        for (int i=0; i< 0) {
            V answer = node.values[i];
            for (int j=node.size; j>i; j--)
            node.keys[j] = node.keys[j-1];
            node.values[j] = node.values[j-1];
        }
        node.keys[i] = key;
        node.values[i] = val;
        return answer;
    }
    else {
        for (int i=0; i<node.size; i++) {
            int comp = key.compareTo(node.keys[i]);
            if (comp <= 0) {
                V answer = put(node.children[i], key, val);
                if (node.keys[i].size == M)
                    split_node(i);
                return answer;
            }
        }
        V answer = put(node.children[node.size], key, val);
        if (node.keys[i].size == M)
            split_node(size)
        return answer;
    }
}

- **Example** In a B-tree with M=6, add

  10 20 30 40 50 60 25 34 70 80 90 35 54 48 49 50 85 32 33 42 72 74 76

- **Which to choose?**

  o Is the dataset relatively small?

    - Will contains be the prominent operation?: Sorted array with binary search
    - Will the application involve adding and deleting data? Balanced binary tree

  o Is dataset many times the size of main memory? (in which case most of it will be in disk memory) B-tree