1 Binary search trees (BSTs)

- Represented as ordinary binary trees
- Maintain binary search tree property – nodes to the left are less than the current node, nodes to the right are greater

2 Binary search tree property

- Given items \( x \) and \( y \), \( x.\text{key} < y.\text{key} \) or \( y.\text{key} < x.\text{key} \)
- Transitive property holds – if \( x.\text{key} < y.\text{key} \) and \( y.\text{key} < z.\text{key} \) then \( x.\text{key} < z.\text{key} \).
- All nodes must have keys by which to compare
- For each node, nodes to the left have smaller keys, nodes to the right have greater keys
- When we say that one item is less than another, typically we mean that the key for that item is less than the other
- The key for a node can be its value

3 Finding elements in a BST

- Intuitive – search to the left if the item we’re looking for is less than the current, search to the right if it’s greater
- Plan – When searching for the item with the key \( k \), if the current node \( T \) has \( T.\text{key} == k \), return it. If \( k < T.\text{key} \), search left recursively. If \( k > T.\text{key} \), search right recursively. If we reach null through recursion, then the item is not in this BST.
- Search at most \( h+1 \) items, where \( h \) is the height of the tree

4 Inserting elements in a BST

- Intuitive – search until the terminal location is found at which the item should be placed, and place it there
- Plan – To insert an item \( n \) into the BST, search the BST as if the item were in the tree, checking to see if the child being recursed on is null. If it is, place \( n \) there
- We’re not dealing with duplicate items in BSTs – if a node with the same key is found, it is replaced
5 Removing elements from a BST

- There are several cases when removing a node \( n \) from a BST
- Leaf nodes – simply find the leaf node and remove it by setting the pointer to it from its parent to null
- Nodes with one child – if node \( n \) is being removed and has parent node \( p \) and child node \( c \), remove node \( n \) by making \( p \)'s child \( c \), taking the place of \( n \)
- Nodes with two children – we need the node that has the value closest to node \( n \) being removed (why?), so we select either \( n \)'s left child's rightmost node or \( n \)'s right child's leftmost node and replace \( n \) with it

6 Balance and completeness

- These operations do not guarantee balance – insertion of items with keys 0, 1, 2, 3, 4, 5 in order will create a “stringy” tree that looks like a linked list
- We want trees to be “bushy”, meaning that they should be reasonably well balanced – complete, or nearly complete
- A complete tree is one in which all leaf nodes occur last if read top-down, left to right – such a tree is maximally balanced
- There are various notions of balance and ways of ensuring it (we’ll see these in later weeks)

7 Array based tree representation

- Place the root node at index 0
- For each node at index \( k \), its left child is at \( 2k+1 \) and its right child is at \( 2k+2 \)
- Works well for complete trees or very well balanced trees (why?)
- Can be used for things like trees

```java
/** Gets the index of the parent of \( n \) in \( h \) */
private int parent(int n) {
    return (n-1)/2;
}

/** Gets the index of the left child of \( n \) in \( h \) */
private int leftChild(int n) {
    return (n*2)+1;
}

/** Gets the index of the right child of \( n \) in \( h \) */
private int rightChild(int n) {
    return (n*2)+2;
}
```
8 Announcements
- Lab 5 is about the project
- Exam tomorrow (Tuesday) in 155 Dwinelle at 6:40 PM
- Homework 3 is out (but much easier)
- Reading (for today, but not on the midterm): Data Structures (Into Java) 5.4, 6; Goodrich & Tamassia 7, 9.1

9 Priority queues
- Queues with ordering based on “priority”, which is some notion of the value of items (which may or may not be the item itself)
- Operations on a priority queue – add, max or min, extractMax or extractMin.
- A priority queue usually either supports max and extractMax or min and extractMin, but not both pairs of operations.

10 Simple priority queue implementation
- Store items in a linked list on add
- Search for biggest item on extractMax or max
- Simple, but slow – must search through the entire linked list to find the max element each time

11 Binary heap priority queue implementation
- Use a binary heap – a tree-like structure that maintains the heap property
- In a max-heap (one that represents a maximum priority queue), the heap property requires that each node in the heap has priority greater than its children
- Node at the root of the tree is guaranteed by the heap property to be the item with the maximum priority
- A binary heap is like a complete binary tree, so we use an array-based tree representation
- Two internal operations maintain the heap property – reHeapifyUp and reHeapifyDown
- Since the heap should be able to grow, can use a vector to be an internal array-based representation

Constructing the binary heap
```java
public class BinaryHeap {
    private Vector h; // the heap

    public BinaryHeap() {
        h = new Vector();
    }

    ...}
```
12 Binary heap insertion

- Insert into the first free index of the array; since the “tree” is complete, the first free index will be after all existing indices
- Insertion will likely violate the heap property (because we may insert a value at a location at which it has a higher priority than its parent), so we need to restore the heap property
- We `reHeapifyUp` – move the inserted item up the heap by swapping it with its parent until it is no longer greater than its parent. This operation may move the item all the way to the root

Inserting into the binary heap

```java
public void insert(Comparable c) {
    h.add(c);
    reHeapifyUp(h.size()-1);
}
```

13 Binary heap extractMax

- Remove the element at the top (root) of the heap – this element is guaranteed to be the maximum
- Replace it with the last element in the heap – this will likely violate the heap property
- Perform `reHeapifyDown` from the top of the heap – move the item we just placed at the top downward by swapping it with its larger child until it is greater than both of its children

```java
public Comparable extractMax() {
    Comparable temp = (Comparable) h.get(0);
    h.setElementAt(h.get(h.size()-1), 0);
    h.remove(h.size()-1);
    if (h.size() > 0) {
        reHeapifyDown(0);
    }
    return temp;
}
```

14 Binary heap reheapify

```java
/** Re-heapifies upwards from n */
private void reHeapifyUp(int n) {
    if (n <= 0)
        return;
    int cur = n;
    while (cur > 0 & &
            ((Comparable) h.get(cur)).compareTo(h.get(parent(cur))) > 0) {
        swap(cur, parent(cur));
        cur = parent(cur);
    }
}
```
/** Re-heapifies downwards from n */
private void reHeapifyDown(int n) {
    int cur = n;
    int size = h.size()-1;
    int largeChild = 0;
    for (;;) {
        /* If we're on leaf nodes, we're done */
        if (leftChild(cur) > size)
            break;
        /* Figure out which of the children is bigger */
        if (rightChild(cur) > size ||
            ((Comparable) h.get(leftChild(cur))).compareTo(h.get(rightChild(cur))) > 0)
            largeChild = leftChild(cur);
        else
            largeChild = rightChild(cur);
        /* Is the largest child larger than cur? */
        if (((Comparable) h.get(cur)).compareTo(h.get(largeChild)) > 0)
            break;
        /* It wasn’t, so swap them */
        swap(cur, largeChild);
        cur = largeChild;
    }
}

15  Question

• Suppose we have an unsorted array of elements from which we want to build a heap.

• We could create a new heap and perform insert operations repeatedly for each item in the array, inserting them into the heap

• Is there a better (faster) way?

• We’ll talk more about this later in the week