Balanced Trees

Chapter 10
Heap

- A heap is a binary tree such that
  - the data contained in each node is greater than (or equal to) the data in that node’s children.
  - the binary tree is complete
Is it a heap?

NOT COMPLETE!
Is it a heap?

NO!
Is it a heap?

YES!
Storage of a heap

• Use an array to hold the data.
• Store the root in position 0.
• For any node in position i,
  - its left child (if any) is in position $2i + 1$
  - its right child (if any) is in position $2i + 2$
  - its parent (if any) is in position $(i - 1)/2$

(integer division)
left child: $2i + 1$
right child: $2i + 2$
parent: $(i - 1)/2$
Basic operations on a heap

- Create an empty heap (constructor).
- Insert a data element into a heap.
- Remove the maximum data element from the heap.
The heap data structure

```java
public class Heap {  
    private int[] data;  
    private int heapSize;  
    private int maxSize;  
    public Heap(int maximumSize) {  
        if (maximumSize < 1) maxSize = 100;  
        else maxSize = maximumSize;  
        data = new int[maxSize];  
        heapSize = 0;  
    }  
    public boolean isEmpty() // not  
    public boolean isFull() // shown
```
Inserting into a heap

• Place the new element in the first available position in the array.
• Compare the new element with its parent. If the new element is greater, than swap it with its parent.
• Continue this process until either
  - the new element’s parent is greater than or equal to the new element, or
  - the new element reaches the root
Inserting into a heap

Insert 38

24 -> 48 -> 38
Inserting into a heap

Insert 70
Inserting into a heap (cont’d)

Insert 70
Inserting into a heap (cont’d)

Insert 70

Tree:
- Root: 95
- Left child of root: 70
  - Left child of 70: 53
  - Right child of 70: 61
    - Left child of 61: 24
    - Right child of 61: 48
    - Right child of 61: 39
  - Right child of 70: 83
    - Left child of 83: 72
    - Right child of 83: 16
Inserting into a heap

```java
public void insert(int item) {
    int position;
    if (isFull()) throw new Exception();
    heapSize++;
    data[heapSize-1] = item;
    position = heapSize - 1;
    while (position > 0 &&
           data[position] > data[(position-1)/2]) {
        swap(position, (position-1)/2);
        position = (position-1) / 2;
    }
}
```
Removing from a heap

• Place the root element in a variable to return later.
• Move the last element in the deepest level to the root (reduce the size of the heap by 1).
• **While** the moved element has a value lower than one of its children, swap this value with the highest-valued child.
• Return the original root that was saved.
Removing from a heap

return value 95
Removing from a heap (cont’d)
Removing from a heap (cont’d)
Removing from a heap
Removing from a heap

public int remove() {
    int answer;
    if (isEmpty()) throw new Exception();
    answer = data[0];
    data[0] = data[heapSize-1];
    heapSize--;
    fixheap();
    return answer;
}
private void fixheap() {
    int position = 0; int childPos;
    while (position*2 + 1 < heapSize) {
        childPos = position*2 + 1;
        if (childPos < heapSize-1 &&
            data[childPos+1] > data[childPos])
            childPos++;
        if (data[position] >= data[childPos])
            return;
        swap(position, childPos);
        position = childPos;
    }
}
Efficiency of heaps

Assume the heap has N nodes.
Then the heap has approximately $\log_2 N$ levels.

- **Insert**
  Since the insert swaps at most once per level, the order of complexity of insert is $O(\log N)$

- **Remove**
  Since the remove swaps at most once per level, the order of complexity of remove is also $O(\log N)$
Priority Queues

• A priority queue PQ is like an ordinary queue except that we can only remove the “maximum” element at any given time (not the “front” element necessarily).

• If we use an array for the PQ, enqueue takes $O(1)$ time but dequeue takes $O(n)$ time.

• If we use a sorted array for the PQ, enqueue takes $O(n)$ time, while dequeue takes $O(1)$ time.

• We can use a heap to implement a priority queue, so that enqueue and dequeue take $O(\log N)$ time.
B-trees

• A B-tree is a balanced search tree.
• A B-tree is designed to work well on magnetic disks and other direct-access secondary storage devices by minimizing disk I/O operations.
• A node can hold more than one item.
• A node can have more than two children.
B-tree rules

• The root may have as few as one element. Every other node has at least MINIMUM elements.

• The maximum number of elements in a node is twice the value of MINIMUM.

• The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (in position 0) to the largest element.
General Idea

$\text{item}_1, \text{item}_2, \ldots, \text{item}_k$

AND

$\text{items} < \text{item}_1$

AND

$\text{items} < \text{item}_2$

AND

$\text{items} < \text{item}_3$

AND

$\text{items} > \text{item}_1$

AND

$\text{items} > \text{item}_2$

AND

$\text{items} > \text{item}_3$

AND

$\text{items} > \text{item}_k$
B-tree rules (cont’d)

• The number of subtrees below a non-leaf node is always one more than the number of elements in the node.
• For any non-leaf node, an element at index $i$ is
  (a) greater than all the elements in subtree $i$ of the node, and
  (b) less than all the elements in subtree $i+1$ of the node
• Every leaf in a B-tree has the same depth.
Sample B-tree
MINIMUM = 2

Insert 44:
Searching in a B-tree

- Searching for an item in the current node:
  - Let $k =$ number of data values in this node
  - Let $i =$ the first index such that $\text{data}[i] \geq \text{item}$. (If every data value in the node is $< \text{item}$, set $i = k$.)
  - If $\text{item} = \text{data}[i]$, return true. (assume $i < k$)
  - Else if node has no children, return false
  - Else call search recursively on subtree[i]
2-3-4-trees

- A 2-3-4 Tree is a tree in which each internal node (nonleaf) has two, three, or four children, and all leaves are at the same depth.
Sample 2-3-4-tree

Insert 100:
Sample 2-3-4-tree

Insert 25:
2-3-4 trees vs. B-trees

- The more data values in a node, the shorter the tree will be, which translates into less searches for an item.
- However, the more data in a node, the greater the number of comparisons to determine the subtree to examine.
- 2-3-4 trees good for trees in main memory.
- B-trees good for external memory searches.
Red-Black Trees

• A red-black tree has the advantages of a 2-3-4 tree but requires less storage.

• Red-black tree rules:
  - Every node is colored either red or black.
  - The root is black.
  - If a node is red, its children must be black.
  - Every path from a node to a null link must contain the same number of black nodes.
Red-Black Trees
Sample Red-Black Tree

Original 2-3-4 tree:

Equivalent red-black tree:
Insert in Red-Black Tree

Insert 85
Sample Red-Black Tree

Insert 80
AVL Trees

• An AVL tree is a **balanced** binary search tree. (AVL comes from the inventors: Adelson, Velski, and Landis)

• The main idea is to maintain that for every node, and its subtrees. Depths differ by no more than 1.

• If the property above is violated, the tree is “rotated” to maintain its balance.

• The depth of an AVL tree with n nodes is $O(\log n)$
AVL Examples
AVL Example

rotate left
Counting Binary Trees

for n=1, count=1
Counting Binary Trees (cont’d)

for $n=2$, count=2
Counting Binary Trees (cont’d)

for $n=3$, count=5
Counting Binary Trees (cont’d)

for n=3, count=5
Counting Binary Trees (cont’d)

for $n=3$, count=5

\[ b_3 = b_0 \times b_2 + b_1 \times b_1 + b_2 \times b_0 = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5 \]
Counting Binary Trees (cont’d)

General Form

\[ b_n = b_0 \cdot b_{n-i-1} + b_1 \cdot b_{n-i-2} + \ldots + b_{n-i-1} \cdot b_0 \]