Heaps

Non-binary trees

October 22 and 24, 2003

Heaps

- Special case of a binary tree
- Parent node $\geq$ its child nodes
  - duplicates are OK
  - left/right ordering is not important here
- Leaves are all as far left as possible
  - i.e., leaves are filled in left to right
  - tree is perfectly balanced
- Height is $O(lg n)$

Min and max heaps

- Previous slide: max heap
  - node with biggest value is the root
- Replace “$\geq$” with “$\leq$” on previous slide: min heap
  - node with smallest value is the root

Heaps and arrays

- A heap can be easily implemented as an array
- Nodes are placed in array sequentially, from top to bottom and left to right (breadth-first traversal)
- A node at position $i$ in the array has its children at positions $2i + 1$ and $2i + 2$
Converting arrays to heaps

- 2 methods:
  - Williams algorithm: top-down
    - idea: new elements are added to leaves, and moved up as necessary
    - worst case: $\lg(n!) \text{ swaps} = \Theta(n \lg n)$
  - Floyd algorithm: bottom-up
    - idea: make small heaps and merge these together
    - worst case: later

Williams algorithm

while not end of array,
  if heap is empty,
    place item at root;
  else,
    place item at bottom of heap;
    while (child > parent)
      swap(parent, child);
    go to next array element;
end

Floyd algorithm

for $i = \text{index of last non-leaf node to 0}$,
  root = array($i$);
  while this tree is not a heap,
    find topmost out-of-order node;
    move this node down;
    repeat until node is correctly placed.

moveDown method

input: array, index of non-leaf node, index of last node
while not end of array,
  find larger child of node;
  if parent is smaller than child,
    swap parent and child;
  set parent = larger child.
Floyd algorithm: worst case complexity

- All nodes on second to last level are moved down one level
  - $(n+1)/4$ swaps
- All nodes on the level above this are moved down to the leaves (2 levels)
  - $2(n+1)/8$ swaps
- ...
- Root is moved all the way down to leaf level
  - $\log(n+1) - 1$ swaps $\Rightarrow \log((n+1)/2))$

At level $i$, make $(i-1)(n+1)/2^i$ swaps

Sum all possibilities from 2 to $\log(n+1)$

$\Rightarrow O(n)$

Note 1: Average case for Williams and Floyd is $O(n)$

Note 2: Book says Williams is better than Floyd for worst case $\Rightarrow$ WRONG

Applications

- Heapsort: later
- Priority queue
  - linked list implementation (previously): searching is $O(n)$
  - heap implementation: searching is $O(\log n)$
  - heap is more efficient implementation

enqueue method

place item at end of heap
while item is not in the root and item is greater than its parent,
swap item with its parent
**dequeue method**

remove element at root
make last leaf the root
remove the last leaf
let p = root
while p is not a leaf and p < either child,
    swap p and larger child

**Multiway trees**

- Each node has multiple keys and multiple children (2 or more)
- Examples:
  - B-Trees
  - Red-black trees
  - Tries

**Multiway tree: definition**

- Each node has \( m \) children and \( m-1 \) keys
- Keys are in ascending order within node
- For node \( i \):
  - keys 0 through \( i-1 \) are less than key \( i \)
  - keys \( i+1 \) through \( m \) are greater than key \( i \)

**Uses of multiway trees**

- Quick searching for information
- Quick updating of information
- Information is stored on physical media
  - disk, tape, etc.
  - \( \text{time(access)} = \text{time(seek)} + \text{latency} + \text{time(transfer)} \)
  - slower than when stored in memory
- \( \Longrightarrow \) Databases
B-Trees

- Ideal when you need to access data on physical media
- Nodes can be tuned to minimize the number of accesses to the media
  - variable size
- Each node stores data *keys* and data *references*

Properties of B-trees

- Root has at least 2 subtrees (unless it is a leaf)
- Non-root, non-leaf nodes store $k-1$ keys and $k$ references to subtrees, $ceil(m/2) \leq k \leq m$
- Leaf nodes store $k-1$ keys, $ceil(m/2) \leq k \leq m$
- All leaves are on the same level

**B-trees are small, perfectly balanced, and always at least half full**

B-tree node: implementation

```java
public class BTreeNode {
    int m = ... // some large number;
    boolean isLeaf = true;
    int numKeys = 1;
    int keys[] = new int[m-1];
    BTreeNode references[] = new BTreeNode[m];
    BTreeNode(int key) {
        keys[0] = key;
        for (int i=0; i<m-1; i++)
            references[i] = null;
    }
}
```

B-tree search

- Time required to search a B-tree is proportional to its height
- Height of a B-tree:
  \[ h \leq \log_q \left( \frac{(n+1)}{2} \right) + 1 \]
  \[ q = ceil(m/2) \]
- B-trees are small!
  - searching is $O(lg n)$
B-tree search: implementation

```java
public BTreeNode search(int key) {
    return search(key, root);
}
```

```java
protected BTreeNode search(int key, BTreeNode node) {
    if (node != null) {
        for (int i=1; i<=node.keyTally && node.keys[i-1]<key; i++);
        if (i > node.keyTally || node.keys[i-1] > key)
            return search(key, node.references[i-1]);
        else
            return node;
    } else
        return null;
}
```

B-tree insertion

- Idea: place key in a leaf, rearrange keys as necessary
- 3 cases:
  - key placed in leaf, no need to rearrange keys
  - shift keys right as necessary
  - key placed in full leaf
    - split the leaf
    - move half the keys to the new leaf
    - move last key of old leaf to parent node
    - place reference to new node in parent node

B-tree insertion (cont.)

- key placed in full leaf, root is full
  - follow procedure for Case 2 to insert key in leaf
  - split root
  - move half the keys to new node
  - create new root
  - move last key from old root to new root
  - create references
  - height of tree increases

How often do nodes split?

- probability = rate at which splits occur
- Tree with \( k \) nodes and height \( h \) splits \( k-h \) times
- Number of keys is at least \( 1 + \lceil m/2 \rceil - 1 \)
- \( P(\text{node splits}) = 1/(\lceil m/2 \rceil - 1) \)
- The larger the capacity of a node, the less likely it is to split
B-tree: deletion

- Key point: nodes have to be at least half-full at all times!
- 2 cases:
  - delete key from leaf: a bit complicated
  - delete key from non-leaf:
    - replace key with predecessor or successor (from leaf)
    - problem now reduces to delete from leaf

B-tree deletion algorithm

```
delete (key):
    node = search(key, root);
    if (node != null), // found the key
        if non-leaf node,
            find leaf with successor of key;
            copy successor into key's slot in node;
            node = successor's node;
            delete successor from node.
        else, delete key from node.
        while (true),
            if no underflow, return.
            else if sibling has room,
                redistribute keys between node and sibling;
                // promote or demote keys as necessary
                return;
        // continued on next slide
    else if node's parent is root,
        if parent has one key,
            merge node, sibling, and parent --> new root.
        else, merge node and sibling.
        return.
    else
        merge node and sibling;
        node = parent.
```

B-trees and space

- B-trees must be at least 50% full at all times
- Typically about 69% full
- Some space is therefore wasted
- Variations on B-trees:
  - B*-trees: each node is at least 2/3rds full
  - B+-trees: all data references are in the leaves
  - red-black trees (a type of 2-4 tree)
Trie

• From the word retrieval (pronounced “try”)
• Useful for performing searches on characters:
  – spell checker
  – dictionary lookup
  – regular expressions

Trie: implementation

• Each node contains a number of letters (between 1 and 26)
• Each letter contains a reference to another node containing letters
• Leaves: letter references point to words (constructed by moving down the tree) or “null” (no word with this combination of letters)