CIS 313

week of Oct 15

fourth week of the term
binomial trees

• a *binomial heap* will be a collection of binomial trees with the heap property

• so we need to define a *binomial tree* first

• a binomial tree is defined recursively:
  • a $B_0$ tree is a single node (height 0)
  • a $B_k$ tree consist of a $B_{k-1}$ tree whose root has another $B_{k-1}$ tree as a child

• a $B_k$ tree contains $2^k$ nodes

• the number of nodes on level $j$ of a $B_k$ tree is the binomial coefficient $\binom{k}{j}$
these trees will be represented using the first-child next sibling representation of ordered trees
binomial heap

• collection of binomial trees
• values stored in the nodes satisfy heap property
• min value could be at root of one of the trees
• if n nodes stored, then \( \lg n \) trees used, corresponds to binary representation of n
• example: if \( n=13 \), need \( B_0, B_2, B_3 \) trees (containing 1, 4, 8 nodes)
• \( \ldots n=13=(1101)_2 \) in base 2
example

- $n=13=(1101)_2$
- a $B_0$, $B_2$, and $B_3$ tree

source:
https://en.wikipedia.org/wiki/Binomial_heap
merge two $B_k$ trees

- two $B_k$ trees can be merged into a $B_{k+1}$ tree
- look at the two roots ...
- ... the root with larger value becomes child of root with smaller value
- easy since children of root given in linked list
- result is $B_{k+1}$ tree
main operation: union of two binomial heaps

• two heaps of sizes n and m can be merged in time $O(\lg n + \lg m)$
• idea is simple:
  • for $k = 0, 1, 2, \ldots$
  • scan through each heap’s tree list
  • if there are two $B_k$ trees, merge them together into a $B_{k+1}$ tree
  • (note 1: one of the $B_k$ trees might be the result of an earlier merge)
  • (note 2: there might be three $B_k$ trees, one each from the two heaps and one from an earlier merge – pick any two of them – similar to a carry bit)
• operation parallels closely addition in binary
example union

\[\begin{array}{c}
1011 \\
\text{+} \\
0011 \\
\hline
1110
\end{array}\]

\[\begin{array}{c}
11 \\
\text{+} \\
3 \\
\hline
14
\end{array}\]
other operations “reduce” to union

• insertion:
  • to insert $x$ into heap $H$
  • create heap $H'$ consisting of only $x$
  • perform union of $H$ and $H'$

• time $O(\log n)$
  • actually not so bad if many insertions performed
  • a sequence of $n$ insertions into an initially empty heap uses $O(n)$ time
  • similar: $n$ increments (by one) of a binary counter (initially zero) makes $O(n)$ bit flips
  • analysis: we saw something like $\sum_{i=0}^{\log n} i2^i = O(n)$ with the BuildHeap routine
extract-min

- the min is the root of one of the trees in the binomial heap H
- suppose it’s a $B_k$ tree with root $x$
  - pull the tree with root $x$ out of $H$
  - the children of $x$ form a binomial heap $H'$
    - ($H'$ will have one each of a $B_0$, $B_1$, ..., $B_{k-1}$ tree)
  - perform a union $H'$ and the reminder of $H$
  - return key of $x$

- $O(lg\ n)$ time
1 is the min of H

pull out the tree with 1 as root

remove 1 and look at its child list as heap H'

get union of H’ with remaining heap H
lazy binomial heaps

- can make insert and union LAZY
  - to perform union of two heaps, just concatenate the tree lists
  - keep pointer to min element
  - may have many $B_k$ trees for some $k$
- only merge $B_k$ trees into $B_{k+1}$ trees when necessary
- extractMin
  - need to scan list to look for new min
  - perform COALESCE operation: for $k=0, 1, 2, ...$ as long as there are at least two $B_k$ trees, merge them
  - might be $O(n)$ time, but can show that is very rare
- good amortized behavior
- leads into FIBONACCI heaps
binary search trees

• chapter 12
• we will look at
  • definitions
  • properties
  • operations: insert, delete, search
  • traversals: inorder, postorder, preorder, level order
  • worst case behavior
  • average case behavior
• then move onto self-balancing BSTs: red-black, 2-3, 2-3-4, ...
various trees

- free tree
- rooted tree
- ordered tree
- binary tree
- binary search tree
  - (search property) let x be a node in a BST. If y is a node in the left subtree of x, then y.key <= x.key. If y is in the right subtree of x, then y.key >= x.key
assorted facts and definitions

• any tree with n nodes has n-1 edges
• a binary tree with left/right pointers and n nodes has n+1 null pointers
• a full binary tree with n internal nodes has n+1 external nodes
• full binary tree: all nodes have either 2 children (the internal nodes) or 0 children (external)
• a binary tree of n nodes has height at least $\lg n$ and at most n-1
• height = distance of node from bottom, depth = distance from top
facts, defs cont’d

• internal path length (I): sum of the depths of all the nodes
• external path length (E): sum of the depths of the nulls (externals)
• fact: \( E = I + 2n \) (nice exercise)
• \( I \) corresponds to successful search in BST, average search time is \( 1 + \frac{l}{n} \)
• \( E \) corresponds to unsuccessful search, average failed search time is \( \frac{E}{n+1} \)
sample BST
BST operations

• find(x)
• insert(x): find a null and put it there
• successor(x)
  • successor(10)=11, successor(15)=17
  • algorithm?
    • if x has right child, go right once, then left until end
    • otherwise, follow parent links until “right” turn
• delete(x): how?
  • if 0 children, remove
  • if 1 child, splice out
  • if 2 children, replace with successor value, then remove successor node
walks

- inorder
  - 1 3 4 5 7 8 9 10 11 12 13 15 17 18 20 23

- preorder
  - 12 10 5 3 1 4 8 7 9 11 17 13 15 20 18 23

- postorder
  - 1 4 3 7 9 8 5 11 10 15 13 18 23 20 17 12