IE 495 Lecture 12

October 5, 2000

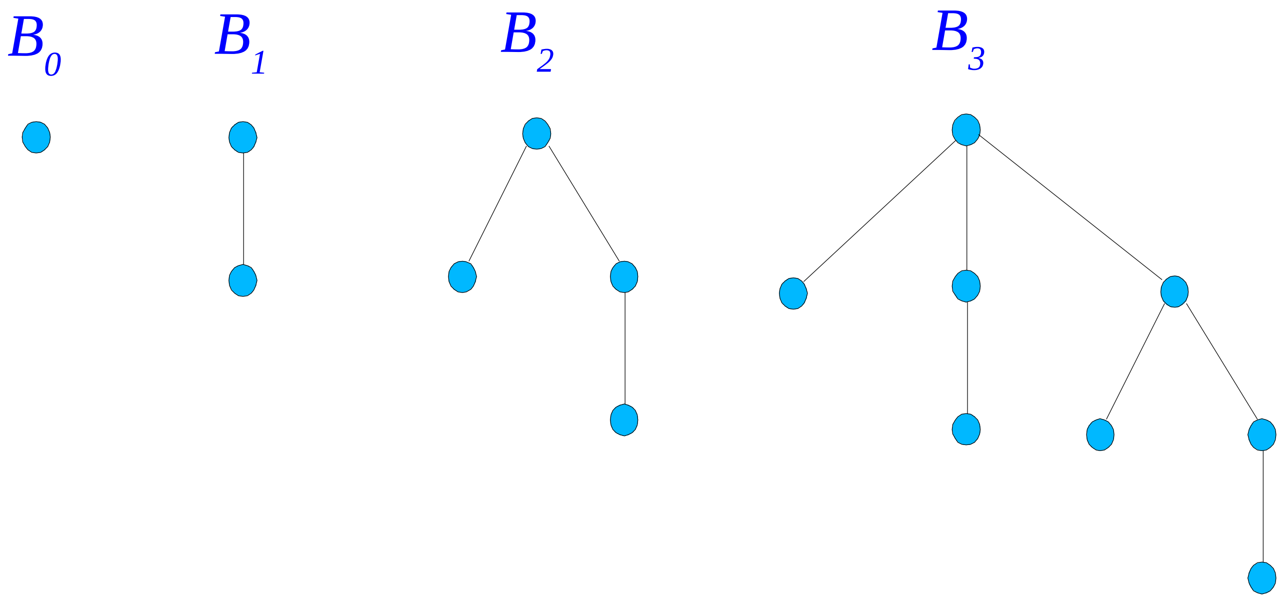
Reading for This Lecture

- Primary
  - Horowitz and Sahni, Chapter 2, Section 3
  - Kozen, Lectures 8-11

Review From Last Time

Binomial Trees

- The *binomial tree* of rank i (B_i) is defined recursively.
- B_i consists of a *root* with i children $B_0, \ldots, B_{i-1}$.



Binomial Heaps

- A *binomial heap* is a collection of heap ordered binomial trees and a pointer to the overall max/min.
- No more than one tree of each rank is allowed.
- The children of each vertex are maintained in a circular linked list.
- The basic operation is *linking*.
- Two trees of rank i can be combined into one tree of rank $i+1$ in constant time.

Eager Meld

- We can combine two heaps by performing a `meld()` reminiscent of binary addition.
- Successively link trees of equal rank and "carry" one if necessary.
- Must track the position of the new min/max element.
- This operation takes $O(\log n)$ time.

Inserting into a Binomial Heap

- To `insert()` an element:
  - Make a new heap from the single element to be inserted.
  - Meld the new heap with the old one.
- To `make_heap()` from scratch, perform a sequence of inserts.
- To `delete()` the min/max element:
  - The children of this element form a new binomial heap.
  - Meld the old heap and the new one.

Amortized Analysis

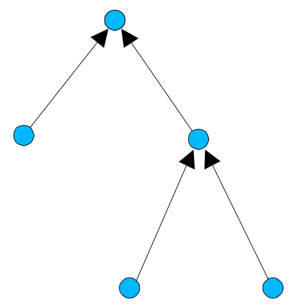
- *`meld()`* and *`delete()`* both take O(log n).
- We will use *amortized analysis* to show that *`insert()`* is constant time overall.
- Idea: The total number of linking operations can never be more than the number of insert operations.
- This means that any sequence of inserts takes constant time *on average*.

Data Structures for Disjoint Sets

- We have a set S and a partition $S_1, \ldots, S_n$ of S.
- We want a data structure that supports
  - union()
  - find()
- Applications
  - Constructing equivalence classes
  - Graph algorithms

Union-Find

- Represent each member of the partition as a rooted tree.
- Choose a designated "representative".
- All other elements are connected to the representative.



First implementation

- union()
  - Point root of set *A* to root of set *B*
- find()
  - Follow the path to the root.
- Analysis

A Tale of Two Heuristics

- How can we improve the complexity of `find()`?
- Heuristic 1
- Heuristic 2

Analysis

- Heuristic 1 guarantees that the depth of each tree is no more than $\lfloor log\ n \rfloor + 1$.
- The proof of this is by induction.
- This implies that `find()` can be performed in *O(log n)*
- Heuristic 2 allows us to perform `find()` in *almost* constant time (amortized).

Ackerman's Function

- Ackerman's function is an extremely fast growing function.
- Definition
  - $A_0(x) = x + 1$
  - $A_{k+1}(x) = A_k^{\,x}(x)$, where $A_k^{\,i+1}(x) = A_k(A_k^{\,i}(x))$
- $A_0(x) = x+1$, $A_1(x) = 2^x$, $A_2(x) = x2^x$, $A_3(x) \geq 2 \uparrow x$
- $A_4(2)$ is greater than the number of particles in the known universe or the number of nanoseconds since the Big Bang (large number).

Inverse Ackerman's Function

- Define $A(k) = A_k(2)$.
- Now define $\alpha(n)$ = smallest k such that $A(k) \geq n$
- $\alpha(n)$ is the inverse Ackerman's function
- $\alpha(n)$ is 4 for all practical purposes.
- Let $T(m,n)$ be the running time of a sequence of $m \geq n$ `find()` operations and n-1 `union()` operations.
- $T(m, n) \in O(\alpha(n)(m+n))$

Hash Tables

- Symbol Table
  - Determine presence of an arbitrary element
  - Allow easy insertion and deletion
- Hashing is an easy and efficient implemetation
- Hash function
  - Maps each possible element into a specified bucket
  - The number of buckets is much less than the number of possible elements
  - Each bucket can store a limited number of elements

Parameters

- T = total number of possible elements
- b = number of buckets
- s = number of elements allowed in each bucket
- n = number of elements in the table
- n/T = element density
- $\alpha = n/sb$ = loading density

Hash Functions

- *Collision*: two elements map to the same bucket
- *Overflow*: too many elements in one bucket
- Choosing a hash function
  - easy to compute
  - minimize collisions
- If $P(f(X) = i) = 1/b$ over all elements X, then f is a *uniform hash function*

Sample Hash Function

- Interpret the element of the set as an integer X
- Take the hash function to

$$f(X) = X \bmod M$$

- M is the number of buckets
- The choice of M is critical
- M should not be a power of 2 or an even number
- M should be a prime number with some other nice properties

Overflow Handling

- Use the next available slot
  - Bad performance when the hash table fills up.
  - Can end up searching the whole table.
  - Average number of comparisons $(2-\alpha)/(2-2\alpha)$.
- Use linked lists
  - Only compare items with same hash value.
  - Average number of comparison $1 + \alpha/2$.
- Average case for hash tables is good, but worst case is very bad.