EECS 477: Introduction to algorithms.

Lecture 9

Prof. Igor Guskov
guskov@eecs.umich.edu

October 3, 2002

Lecture outline: data structures (chapter 5)

- Arrays, STL vectors
- Graphs
- Trees, balanced trees
- Heaps, binomial heaps
- Associative tables, hashing
- Disjoint subsets (union-find)
- Sets: red-black trees vs. sorted arrays

Arrays

- `float x[n]`: fixed number of elements
- access via indices, constant time address calculation, read/write $O(1)$ – elementary operation
- insertion, maximum value, initialization $\Theta(n)$
- virtual initialization allows initialize while using the array

Arrays: virtual initialization

- requires two auxiliary arrays and a counter
-
```
IDataT& operator[](unsigned i) {
    if(!is_initialized(i))
        init(i);
    return m_px[i];
}

IDataT* m_px;
unsigned *m_pa, *m_pb;
unsigned m_ctr;
```

Arrays: virtual initialization

- `m_pa` stores initialized indices in order
- `m_pb` stores order index at the location
-
```
bool is_initialized(unsigned i) const {
    if(m_pb[i]<m_ctr) {
        if(m_pa[m_pb[i]]!=i) return false;
        else return true;
    } else return false;
}

void init(unsigned i) {
    m_pb[i] = m_ctr;
    m_pa[m_ctr] = i;
    ++m_ctr;
}
```

Arrays: lists

- Records with pointers
- singly linked, double linked, circular
- insertion, removal, successor $O(1)$
- search, min, max $O(n)$

Graphs

- $G = (V, E)$
- V - set of vertices, E - set of edges
- Ex: $(\{1,2,3,4\},\{(1,2),(3,4),(2,3)\})$
- Directed and undirected
- Connected, directed graphs may be strongly connected
- Paths, cycles
- Undirected acyclic graphs: forests (each connected component is a tree)
- Adjacency matrix or list of neighbors representation

Trees

- Rooted trees have a special root node
- Draw with root at the top, children going downwards like a family tree
- Nodes: parents, children, siblings, ancestors, descendants (reflexive!)
- Leaf has no children, others are internal
- Binary trees have no more than two children (k-ary trees)
- Search trees (binary: left children less or equal, right children greater of equal)

Trees

- Nodes have height, depth, and level
  - `height(v) = if leaf(v) then 0 else max(height(children(v)))+1`
  - `depth(v) = if root(v) then 0 else depth(parent(v))+1`
  - `level(v) = height(root(v)) - depth(v)`
- trees have height:

  `height(tree) = height(root(v))`
- Balanced trees have height $= O(\log n)$ where n is number of nodes, then the search is efficient (red-black trees, 2-3 trees, splay trees)

Trees

- Balanced trees are needed for logarithmic search operations
- Red-black and 2-3 trees store additional information
- Rotation: tool for balancing
- Splay trees do not store any additional information – they are self-adjusting
- Will not cover them now, may have a homework problem on that

Projects

- First draft and teams – next week Friday night October 11th
- Final version submit by October 29th
- Approval deadline is October 31st
- Default project will be assigned to everybody else
- Project due December 9th

Projects: list

- Rectangle/box packing
- Power graph coloring
- Covering with disks or boxes
- Clustering
- Scheduling: job interval selection
- Pushing blocks puzzles
- Traveling salesman? Cliques in the graph?

Associative tables

- Keys do not form continuous range like integers, rather sparse like strings
- Symbol table in compilers
- Hash function $h : Keys \to \{0, 1, \ldots, N-1\}$
- When $x \neq y$ but $h(x) = h(y)$ we have a *collision*: resolve it by list chaining
- m - the number of keys, N - allocated array size, then m/N is the *load factor*
- When load factor is below one, access is efficient, otherwise we can rehash doubling the range
- Rehashing helps to maintain constant amortized expected time

Heaps

- Do not confuse with Binary Search Trees!!!
- Rooted tree – no pointers: i is the parent of $2i+1$ and $2i+2$
- Picture: essentially complete binary tree – every internal node has two children except for a special one which may only have the left one
- Heap property: $value(i) \leq value(parent(i))$ for all non-root nodes i
- alter heap: new value higher – percolate, lower – sift down
- make heap: starting from the last sift down – linear time algorithm
- heap sort: $O(n \log n)$ algorithm

Heaps: alter heap

- $O(\log n)$ operations, preserve heap property
- ```
  void sift_down(data* p, int i, unsigned N) {
      while ( i is internal AND key(i)<key(child(i)) ) {
          exchange i with the larger child of i
      }
  }

  void percolate(data* p, int i, unsigned N) {
      while ( i is not root AND key(i)>key(parent(i)) ) {
          exchange i with its parent
      }
  }
  ```
  ```

Heaps: operations

- ```
  void find_max(data* p, int i, unsigned N) {
      return p[0];
  }
  ```

- ```
  void pop_max(data* p, unsigned N) {
      p[0] = p[N-1];
      --N;
      sift_down(p, 0);
  }
  ```

- ```
  void insert(data* p, data d, unsigned N) {
      ++N;
      p[N] = d;
      percolate(p, N);
  }
  ```
  ```

Make-heap: linear algorithm

- ```
  void make_heap(data* p, unsigned N) {
      for(unsigned k = N/2; k>=0; --k)
          sift_down(p, k, N);
  }
  ```
- on level s we have 2^{K-s} nodes each takes s to sift down
- $\sum_{s=1}^{K} s2^{K-s} = O(2^K)$, $N = 2^K$
- Heap property built from leafs to root
  ```

Heap sort

- ```
  void heap_sort(data* p, unsigned N) {
      make_heap(p, N);
      for(unsigned k = N-1; k>=0; --k) {
          swap p[0] and p[N-1];
          sift_down(p, 0, k);
      }
  }
  ```
- $t(N) = O(N \log N)$
  ```

Next time

- binomial heaps and disjoint subsets