Lecture outline: data structures (chapter 5)

• Heaps, binomial heaps
• Disjoint subsets (union-find)
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: $\text{value}(i) \leq \text{value}(\text{parent}(i))$ 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

- \( \sum_{s=1}^{K} s2^{K-s} = O(2^K), \quad N = 2^K \)

- 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);
  }
}

- \( t(N) = O(N \log N) \)
Heap misc

• Sometimes useful to store order of elements in the heap explicitly: define as the inverse function to the heap array.
• This does not change overall asymptotic performance
• To figure out who to percolate and sift-down
• Do not need it if only priority queue without updates is needed.
• $k$-ary heaps make the tree shallower.
# Heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary (wc)</th>
<th>Binomial (wc)</th>
<th>Fibonacci (am)</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-heap</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>find-max</td>
<td>$\Theta(1)$</td>
<td>$O(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>delete-max</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>merge</td>
<td>$\Theta(n)$</td>
<td>$O(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>increase-key</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>delete</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

*
Disjoint sets: union-find

- $N$ objects, set $\{0, 1, \ldots, N - 1\}$
- Partition into disjoint sets – each element in exactly one set
- Each set has a label – one of its members, e.g. the smallest one: $\{2, 3, 7, 9\}$ is “set 2”
- Two operations:
  - FIND: given an object find which set contains it and return its label
  - MERGE(UNION): given two different labels merge the corresponding two subsets
- hence the name: Union-Find data structure
Simple representation

- Array set[N] stores element labels
- $\Theta(1)$ find:
  ```
  int find_simple(int x) { return set[x]; }
  ```
- $\Theta(N)$ merge:
  ```
  void merge_simple(int a, int b) {
    for(k=0; k<N; ++k)
      if(set[k]==max(a,b))
        set[k] = min(a,b);
  }
  ```
Amortized setting

- We would like to perform $n$ finds, and $N - 1$ merges not clear in which order, then the simple algorithms give $\Theta(n)$ for all finds, and $\Theta(N^2)$ for all merges.

- Rooted tree rep: Array $\text{set}[N]$ will store forest of rooted trees, where $\text{set}[i] == i$ would indicate a root node, and otherwise $\text{set}[k]$ gives the index of the parent of $k$.

- Find will follow the parent links to the label node.

- Merge will need to merge two trees.
Rooted tree rep functions

- $\Theta(\text{height})$ find

```c
int find_rooted(int x) {
    while(set[x]!=x)
        x = set[x];
    return x;
}
```

- $\Theta(1)$ merge

```c
void merge_rooted(int a, int b) {
    if(a<b)
        set[b] = a;
    else
        set[a] = b;
}
```
Tree height control

- The above procedure may grow tree height: consider the following sequence on \{0, \ldots, 7\}
do m(6,7), m(5,6), \ldots, m(0,1)

- introduce another array height[N]

- void merge_rank(int a, int b) {
  if(height(a)==height[b]) {
    height(a) = height(b);
    set[b] = a;
  } else {
    if(height(a)<height[b]) set[a] = b;
    else set[b] = a;
  }
}
Tree height control

• **Theorem:** The above merge procedure ensures that after an arbitrary sequence of merges starting from initial situation we have that $\text{height}[a]$ is at most $\lceil \log k \rceil$ where $k$ is the number of nodes in $\text{tree}(a)$.

• **Basis:** initially, zero height everywhere

• **Induction:** assume for $m$ satisfying $1 \leq m < k$. Merge two smaller trees. Let $a \leq b$ and $k = a + b$. Then $a \leq k/2$ and $b \leq k - 1$.
  
  • $h_a \neq h_b$: then $h_k \leq \max(\lceil \log a \rceil, \lceil \log b \rceil)$.
  
  • $h_a = h_b$: then $h_k = h_a + 1 \leq \lceil a \rceil + 1$. But $\lfloor a \rfloor \leq \lfloor \log(k/2) \rfloor \leq \lfloor \log k - 1 \rfloor = \lfloor \log k \rfloor - 1.$
Path compression

- When doing `find` relink all the pointers to the root of the tree: reducing its height

- `int find_compress(int x) {
    int r = x;
    while( set[r]!=r )
        r = set[r];
    while( x!=r ) {
        int j = set[x];
        set[x] = r;
        set[x] = r;
        x = j;
    }
}


Disjoint set structure

- `merge_rank` and `find_compress` form the basis for union-find structure
- `rank` is the upper bound on the height of the tree – because of path compression
- Ackermann’s function variant

\[
A(i, j) = \begin{cases} 
2j, & \text{if } i = 0 \\
2, & \text{if } j = 1 \\
A(i - 1, A(i, j - 1)), & \text{otherwise}
\end{cases}
\]
Disjoint set structure

- $A(1,j) = 2^j$, $A(2,j)$ is $j$ powers of 2 stacked $A(2,4) = 65,536$, grows extremely fast
- $\alpha(i,j) = \min \{ k | k \geq 1 \text{ and } A(k,4 \lceil i/j \rceil) > \log j \}$.
- for all practical purposes $\alpha(i,j) \leq 3$
- Tarjan showed that a sequence of $n$ finds and $m$ merges can be executed in a time in $\Theta((m+n)\alpha(m+n,N))$ where $N$ is the size of the set.
Dijkstra algorithm

- Given a directed graph $G = (N, A)$, $N$ - nodes, $A$ - directed edges (arrows)
- Each edge has non-negative length $L : A \rightarrow \mathbb{R}^+$
- One node is source node
- Find the length of the shortest path from the source to each of the nodes
- Analysis: book and homework
Dijkstra algorithm

void Dijkstra(LengthFn& L) {
    vector<float> D(n); // n nodes
    set<int> C = {1,2, ..., n-1};
    for(i=1; i<n; ++i)
        D[i] = L(1,i)
    for(i=0; i<n-2; ++i) {
        v = find_min( heap on D );
        C.remove(v);
        for_all( w from C )
            D[w] = min( D[w], D[v+L(v,w)] );
    }
}