CSCI-1200 Data Structures — Fall 2015
Lecture 23 – Priority Queues, part II (& Functors)

Review from Lecture 22

- What's a Priority Queue?
- Definition of a Binary Heap: A binary tree where:
  - The value at each internal node, p, is less than the value stored at either of p's children.
  - A *complete* binary tree is one that is completely filled, except perhaps at the lowest level, and at the lowest level all leaf nodes are as far to the left as possible.
- Implementing Pop (a.k.a. Delete Min)
  - The top (root) of the tree is removed.
  - It is replaced by the value stored in the *last leaf node* (how do we find this node? not answered yet!)
  - The last leaf node is removed.
  - `percolate_down` function is then run (starting at the root) to restore the heap property:

```
percolate_down(TreeNode<T> * p) {
  while (p->left) {
    TreeNode<T>* child;
    if (p->right && p->right->value < p->left->value)
      child = p->right;
    else
      child = p->left;
    if (child->value < p->value) {
      swap(child, p);
      p = child;
    }
    else
      break;
  }
}
```

- Implementing Push / Insert
  - A new last leaf node with the new value is added to the tree. `percolate_up` is then run starting at the new node.

```
percolate_up(TreeNode<T> * p) {
  while (p->parent)
    if (p->value < p->parent->value) {
      swap(p, parent);
      p = p->parent;
    }
    else
      break;
}
```

Today's Lecture

- A Heap as a Vector
- Building a Heap
- Heap Sort
- Merging heaps are the motivation for *leftist heaps*
- Something weird & cool in C++... Function Objects, a.k.a. *Functors* (we'll use them next week)

23.1 Warmpup Exercise

Suppose the following operations are applied to an initially empty binary heap of integers. Show the resulting heap after each `delete_min` operation. (Remember, the tree must be **complete**!)

```
push 5, push 3, push 8, push 10, push 1, push 6,
pop,
push 14, push 2, push 4, push 7,
pop,
pop,
pop
```

23.2 Analysis Review

- Both `percolate_down` and `percolate_up` are $O(\log n)$ in the worst-case. Why?

- But, `percolate_up` (and as a result `push`) can be $O(1)$ in the average case. Why?

23.3 Implementing a Heap with a Vector (instead of Nodes & Pointers)

- In the vector implementation, the tree is never explicitly constructed. Instead the heap is stored as a vector, and the child and parent "pointers" can be implicitly calculated.
- To do this, number the nodes in the tree starting with 0 first by level (top to bottom) and then scanning across each row (left to right). These are the vector indices. Place the values in a vector in this order.
- As a result, for each subscript, i,
  - The parent, if it exists, is at location $\lfloor (i-1)/2 \rfloor$.
  - The left child, if it exists, is at location $2i+1$.
  - The right child, if it exists, is at location $2i+2$.
- For a binary heap containing n values, the last leaf is at location $n-1$ in the vector and the last internal (non-leaf) node is at location $\lfloor (n-1)/2 \rfloor$.
- The standard library (STL) `priority_queue` is implemented as a binary heap.

23.4 Exercise

Draw a binary heap with values: 52 13 48 7 32 40 18 25 4, first as a tree of nodes & pointers, then in vector representation.

23.5 Exercise

Show the vector contents for the binary heap after each `delete_min` operation.

```
push 8, push 12, push 7, push 5, push 17, push 1,
pop,
push 6, push 22, push 14, push 9,
pop,
pop,
```

23.6 Building A Heap

- In order to build a heap from a vector of values, for each index from $\lfloor (n-1)/2 \rfloor$ down to 0, run `percolate_down`. Show that this fully organizes the data as a heap and requires at most $O(n)$ operations.

- If instead, we ran `percolate_up` from each index starting at index 0 through index n-1, we would get properly organized heap data, but incur a $O(n \log n)$ cost. Why?

23.7 Heap Sort

- Heap Sort is a simple algorithm to sort a vector of values: build a heap and then run n consecutive `pop` operations, storing each "popped" value in a new vector.
- It is straightforward to show that this requires $O(n \log n)$ time.
- **Exercise:** Implement an *in-place* heap sort. An in-place algorithm uses only the memory holding the input data – a separate large temporary vector is not needed.

23.8 Summary Notes about Vector-Based Priority Queues

- Priority queues are conceptually similar to queues, but the order in which values / entries are removed ("popped") depends on a priority.
- Heaps, which are conceptually a binary tree but are implemented in a vector, are the data structure of choice for a priority queue.
- In some applications, the priority of an entry may change while the entry is in the priority queue. This requires that there be "hooks" (usually in the form of indices) into the internal structure of the priority queue. This is an implementation detail we have not discussed.

23.9 Leftist Heaps — Overview

- Our goal is to be able to merge two heaps in $O(\log n)$ time, where n is the number of values stored in the larger of the two heaps.
  - Merging two binary heaps (where every row but possibly the last is full) requires $O(n)$ time
- Leftist heaps are binary trees where we deliberately attempt to eliminate any balance.
  - Why? Well, consider the most *unbalanced* tree structure possible. If the data also maintains the heap property, we essentially have a sorted linked list.
- Leftists heaps are implemented explicitly as trees (rather than vectors).

23.10 Leftist Heaps — Mathematical Background

- **Definition:** The *null path length* (NPL) of a tree node is the length of the shortest path to a node with 0 children or 1 child. The NPL of a leaf is 0. The NPL of a NULL pointer is -1.
- **Definition:** A *leftist tree* is a binary tree where at each node the null path length of the left child is greater than or equal to the null path length of the right child.
- **Definition:** The *right path* of a node (e.g. the root) is obtained by following right children until a NULL child is reached. In a leftist tree, the right path of a node is at least as short as any other path to a NULL child. The right child of each node has the lower null path length.
- **Theorem:** A leftist tree with $r > 0$ nodes on its right path has at least $2^r - 1$ nodes.
  - This can be proven by induction on r.
- **Corollary:** A leftist tree with n nodes has a right path length of at most $\lfloor \log(n+1) \rfloor = O(\log n)$ nodes.

- **Definition:** A *leftist heap* is a leftist tree where the value stored at any node is less than or equal to the value stored at either of its children.

23.11 Leftist Heap Operations

- The `push`/`insert` and `pop`/`delete_min` operations will depend on the `merge` operation.
- Here is the fundamental idea behind the merge operation. Given two leftist heaps, with `h1` and `h2` pointers to their root nodes, and suppose `h1->value <= h2->value`. Recursively merge `h1->right` with `h2`, making the resulting heap `h1->right`.
- When the leftist property is violated at a tree node involved in the merge, the left and right children of this node are swapped. This is enough to guarantee the leftist property of the resulting tree.
- `Merge` requires $O(\log n + \log m)$ time, where m and n are the numbers of nodes stored in the two heaps, because it works on the right path at all times.

23.12 Merge Code

Here are the two functions used to implement leftist heap merge operations. Function `merge` is the driver. Function `merge_helper` does most of the work. These functions call each other recursively.

```
template <class T>
LeftNode<T>* merge(LeftNode<T> *H1,LeftNode<T> *H2) {
  if (!h1)
    return h2;
  else if (!h2)
    return h1;
  else if (h2->value > h1->value)
    return merge_helper(h1, h2);
  else
    return merge_helper(h2, h1);
}

template <class T>
LeftNode<T>* merge_helper(LeftNode<T> *h1, LeftNode<T> *h2) {
  if (h1->left == NULL)
    h1->left = h2;
  else {
    h1->right = merge(h1->right, h2);
    if(h1->left->npl < h1->right->npl)
      swap(h1->left, h1->right);
    h1->npl = h1->right->npl + 1;
  }
  return h1;
}
```

23.13 Exercises

1. Explain how `merge` can be used to implement `insert` and `delete_min`, and then write code to do so.

2. Show the state of a leftist heap at the end of:

```
insert 1, 2, 3, 4, 5, 6
delete_min
insert 7, 8
delete_min
delete_min
```

23.14 Using STL's for_each

- First, here's a tiny helper function:

```
void float_print (float f) {
  std::cout << f << std::endl;
}
```

- Let's make an STL vector of floats:

```
std::vector<float> my_data;
my_data.push_back(3.14);
my_data.push_back(1.41);
my_data.push_back(6.02);
my_data.push_back(2.71);
```

- Now we can write a loop to print out all the data in our vector:

```
std::vector<float>::iterator itr;
for (itr = my_data.begin(); itr != my_data.end(); itr++) {
  float_print(*itr);
}
```

- Alternatively we can use it with STL's `for_each` function to visit and print each element:

```
std::for_each(my_data.begin(), my_data.end(), float_print);
```

  Wow! That's alot less to type. Can I stop using regular `for` and `while` loops altogether?

- We can actually also do the same thing without creating & explicitly naming the `float_print` function. We create an *anonymous function* using *lambda*:

```
std::for_each(my_data.begin(), my_data.end(), [](float f){ std::cout << f << std::end; });
```

  Lambda is new to the C++ language (part of C++11). But lambda is a core piece of many classic, older programming languages including Lisp and Scheme. Python lambdas and Perl anonymous subroutines are similar. (In fact lambda dates back to the 1930's, before the first computers were built!) You'll learn more about lambda more in later courses like CSCI 4430 Programming Languages!

23.15 Function Objects, a.k.a. *Functors*

- In addition to the basic mathematical operators `+ - * / < >` , another operator we can overload for our C++ classes is the *function call operator*.

  Why do we want to do this? This allows instances or objects of our class, to be used like functions. It's weird but powerful.

- Here's the basic syntax. Any specific number of arguments can be used.

```
class my_class_name {
public:
  // ... normal class stuff ...
  my_return_type operator() ( /* my list of args */ );
};
```

23.16 Why are Functors Useful?

- One example is the default 3rd argument for `std::sort`. We know that by default STL's sort routines will use the less than comparison function for the type stored inside the container. How exactly do they do that?

- First let's define another tiny helper function:

```
bool float_less(float x, float y) {
  return x < y;
}
```

- Remember how we can sort the `my_data` vector defined above using our own homemade comparison function for sorting:

```
std::sort(my_data.begin(),my_data.end(),float_less);
```

  If we don't specify a 3rd argument:

```
std::sort(my_data.begin(),my_data.end());
```

  This is what STL does by default:

```
std::sort(my_data.begin(),my_data.end(),std::less<float>());
```

- What is `std::less`? It's a templated class. Above we have called the default constructor to make an instance of that class. Then, that instance/object can be used like it's a function. Weird!

- How does it do that? `std::less` is a teeny tiny class that just contains the overloaded function call operator.

```
template <class T>
class less {
public:
  bool operator() (const T& x, const T& y) const { return x < y; }
};
```

  You can use this instance/object/functor as a function that expects exactly two arguments of type `T` (in this example `float`) that returns a bool. That's exactly what we need for `std::sort`! This ultimately does the same thing as our tiny helper homemade compare function!

23.17 Another more Complicated Functor Example

- Constructors of function objects can be used to specify *internal data* for the functor that can then be used during computation of the function call operator! For example:

```
class between_values {
private:
  float low, high;
public:
  between_values(float l, float h) : low(l), high(h) {}
  bool operator() (float val) { return low <= val && val <= high; }
};
```

- The range between `low` & `high` is specified when a functor/an instance of this class is created. We might have multiple different instances of the `between_values` functor, each with their own range. Later, when the functor is used, the query value will be passed in as an argument. The function call operator accepts that single argument `val` and compares against the internal data `low` & `high`.

- This can be used in combination with STL's `find_if` construct. For example:

```
between_values two_and_four(2,4);

if (std::find_if(my_data.begin(), my_data.end(), two_and_four) != my_data.end()) {
  std::cout << "Found a value greater than 2 & less than 4!" << std::endl;
}
```

- Alternatively, we could create the functor without giving it a variable name. And in the use below we also capture the return value to print out the first item in the vector inside this range. Note that it does not print all values in the range.

```
std::vector<float>::iterator itr;
itr = std::find_if(my_data.begin(), my_data.end(), between_values(2,4));
if (itr != my_data.end()) {
  std::cout << "my_data contains " << *itr
            << ", a value greater than 2 & less than 4!" << std::endl;
}
```