Justin Ciocoi

Dec. 13, 2023

CSCI 373 Textbook Notes

Chapter 8: Heaps and Priority Queues

8.1: The Priority Queue Abstract Data Type

- A *priority queue* is an abstract data type for stoting a collection of prioritized elements that supports arbitrary element insertion but supports removal of elements in order of priority
- This ADT is fundamentally different from the position-based data structures
- The priority queue stores elements according to their priorities and has no external notion of value *position*
- **8.1.1: Keys, Priorities, and Total Order Relations**
  - Formally, we will define a *key* as an object which is assigned to each object in a collection as a specific attribute for that object and can be used to identify, rank, or weigh that element
  - Each element does not necessarily have a unique key, and an application may even *change* an element's key in order to achieve the desired program
  - A priority queue needs a comparison rule that never contradicts itself
  - For this, we must define a (total order relation) which is to say that the comparison rule is defined for every pair of keys and it must satisfy the following properties
    - *Reflexive Property*: $k \leq k$
    - *Antisymmetric Property*: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
    - *Transitive Property*: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$
  - If these three rules are satisfied, then the comparison rule will never lead to a comparison contradiction
  - A priority queue is a container of elements, each associated with a key

- The fundamental functions of a priority queue, P, are as follows
  - `insert(e)` : Insert the element, e, (with implicit associated key values) into P
  - `min()` : return an element of P with the smallest key value
  - `removeMin()` : Remove the element `min()` from P
- **8.1.2: Comparators**
  - A comparator implemented within a priority queue will have be in the following form:
    - `isLess(a, b)` : where `true` will be returned if $a < b$ and `false` otherwise
- **8.1.3: The Priority Queue ADT**
  - As an ADT, a priority queue P supports the following functions:
    - `size()` : Return the number of elements in P
    - `empty()` : Return `true` if P is empty and `false` otherwise
    - `insert(e)` : Insert the element, e, (with implicit associated key values) into P
    - `min()` : return an element of P with the smallest key value
    - `removeMin()` : Remove the element `min()` from P

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	*"error"*	{}

- **8.1.4: A C++ Priority Queue Interface**

```
template <typename E, typename C>
class PriorityQueue
{
    public:
        int size() const;
        bool isEmpty() const;
        void insert(const E& e);
        const E& min() const;
        void removeMin();
};
```

- **8.1.5: Sorting with a Priority Queue**
    - One very important application of a priority queue is sorting
    - To implement priority queue sorting with a list L of unordered elements and a priority queue P, the following phases will be implemented
        1. Put the elements of L into an initially empty priority queue, P through a series of n insert operations, one for each element
        2. Extract the elements from P in non-decreasing order by a series of n combinations of `min()` and `removeMin()` operations, putting them back into L in order

8.3: Heaps

- An efficient realization of a priority queue uses a data structure called a *heap*, which allows us to perform both insertions and removals in logarithmic time
- A heap will do this by generally abandoning the idea of storing elements and keys in a list and opting to instead store elements and keys in a binary tree
- **8.3.1: The Heap Data Structure**
    - *Complete Binary Tree*: A heap, T with height h is a complete binary tree if levels $\{0, 1, 2, ..., h-1\}$ have the maximum number of nodes and the nodes at level h fill this level from left to right
    - A heap, T, storing n entries has a height:

$$h = [\log n]$$

- **8.3.2: Complete Binary Trees and Their Representation**

- A complete binary tree, T supports all the functions of the binary tree ADT, plus the following two functions:
  - `add(e)`: Add to T and return a new external node v storing element e such that the resulting tree is a complete binary tree with last node v
  - `remove()`: Remove the last node of T and return its element
- For adding, there are essentially two cases to consider
  - If the bottom level of T is not full, then `add()` will insert a new node on the bottom level of T immediately after the rightmost node at this level
  - If the bottom level is full, then `add()` will insert a new node as the left child of the leftmost node of the bottom level of T
- *A vector based representation*
  - For a complete binary tree, T, stored in a vector A such that node v in T is the element of A with an index of $f(v)$ defined by the following rules:
    - If v is the root of T, then $f(v) = 1$
    - If v is the left child of node u, then $f(v) = 2f(u)$
    - If v is the right child of node u, then $f(v) = 2f(u) + 1$
- This can be achieved in C++ using the following interface

```cpp
template <typename E>
class CompleteTree
{
public:
    class Position;
    int size() const;
    Position left(const Position& p);
    Position right(const Position& p);
    Position parent(const Position& p);
    bool hasLeft(const Position& p) const;
    bool hasRight(const Position& p) const;
    bool isRoot(const Position& p) const;
    Position root();
    Position last();
    void addLast(const E& e);
    void removeLast();
    void swap(const Position& p, const Position& q);

    typedef typename std::vector<E>::iterator Position;

private:
    std::vector<E> V;

protected:
    Position pos(int i)
    {
        return V.begin()+i;
    }

    int idx(const Position& p) const
    {
        return p-V.begin();
    }
}
```