Chapter 6
Priority Queues

Introduction

- Want to delete from queue according to priority.
  - Max priority queue – delete the greatest.
  - Min priority queue – delete the least.
- Insert normally, but delete based on priority.
- We can implement priority queues using binary search trees, ordered or unordered lists, ordered or unordered arrays, etc.
- Let assume a linked list implementation.
  - Unordered
    - Insert – $O(1)$.
    - Delete – $O(n)$.
  - Ordered
    - Insert – $O(n)$.
    - Delete – $O(1)$.
- What if we used a BST? What would happen with successive deletes?
- A splay tree? What would happen with successive deletes? Is there a way of getting the good run time without having to have the expense of pointers?

Heaps

- Heap – complete binary tree in which each node is smaller than its parent.
  - Is this the same thing as a binary search tree (BST)?
- The binary tree for the heap is implemented as an array. This allows us easy access to children and parents as seen in the previous chapter.
- Used as a priority queue – regular insertion, priority deletion.
- Insertion – put the new node at the first empty position, and sift the element up (if needed). See Figure 1.
  - $O(\text{height}) = O(\log n)$

![Figure 1 Insertion into a Heap](image-url)
• **Deletion** – select root. Swap with last position (which will no longer be part of queue), and sift-down. See Figure 2.
  - O(height) = O(log n)

![Figure 2 Deletion from Heap](image)

- **Initialization** – insert n items into tree O(n log n) worst case BUT experiments have shown that on average, it only moves up an element 1.6 levels.
- This is an **implicit data structure** – no special structure need for heap except the binary tree. Thus no space overhead.
- In this algorithm, we need to be able to find a parent or a child (based on subscript).
- The book assumes the array begins at position 1, as the math is slightly easier. Using zero based addressing, the formula would be
  - Children(i) = 2i+1 and 2i+2
  - Parent(i) = (i-1)/2
- NOTE: if we have a min heap, we can’t easily find the max or find any particular node.
- The operations decreaseKey, increaseKey, and removeAtLocation can be done fairly quickly if you know the location of the item in the queue.
  - Finding an item in a queue is very time consuming as there is no order.
  - Thus, a decreaseKey operation would need to keep track of the location of items separately.

Skip section 6.4

**d-Heaps**

- Like a heap (stored in an array) but has d kids.
- Shallower. Good for insertion. \( \log_d n \)
- Worse for deletion as have to look at all kids. \( d \log_d n \)
- Division/multiplication by \( d \) is worse (to find parents/kids).
  - Better if \( d \) is a power of 2 so shifting works.
- **Note:** for all heaps we have discussed, merging is bad.
6.6 Leftist Heaps

- To this point, we have focused on the height of a tree
- Define the null path length of a binary tree to be the shortest path from the node to a node without two children
- Define a leftist heap to be one where:
  - both sub-trees are leftist heaps, and
  - the left sub-tree has a null path length greater than or equal to that of the right sub-tree
- The definition suggests a bias to the left, however, this should not suggest that the tree is necessarily left-heavy
- These examples use a min heap – but the idea is the same for a max heap.
- The following are leftist trees (null path length in red):

![Leftist Trees](image)

- All leftist heaps with 1, 2, or 3 nodes:

![Leftist Heaps with 1, 2, or 3 Nodes](image)

- All leftist heaps with four nodes:

![Leftist Heaps with Four Nodes](image)

- Three of the four have more nodes to the left
• As there is no relation between the nodes in the sub-trees of a heap:
  o If both the left and right sub-trees are leftist heaps but the root does not form a leftist heap,
    We only need to swap the two sub-trees
  o We can use this to merge two leftist heaps

• Merging strategy:
  – Given two leftist heaps, recursively merge the larger value with the right sub-heap of the root
  – Traversing back to the root, swap trees to maintain the leftist heap property

Node * Merge (Node * t1, Node *& t2) // t1 and t2 are merged, yielding t1
{     if (t1==NULL)  return t2;
if (t2==NULL) return t1;
if (t1 ->element < t2->element)
    { t1->right = merge(t1->right, t2);
      root=t1;
    }
else
    {
      t2->right = merge(t2->right, t1);
      root=t2
    }
if (notLeftist(root)) swapkids(root);
setLeftist(root);
return root;
}

• Dequeuing strategy:
  – remove the top node and merge the two sub-trees together

• Consider merging these two leftist min heaps

• Comparing 3 and 4, we exchange the two heaps and merge the detached sub-heap with the right sub-heap of 3
• Comparing 4 and 5, we exchange the two heaps and merge the detached sub-heap with the right sub-heap of 4

![Diagram]

• The right sub-heap of 4 is empty, and therefore we attach the heap with root 5

![Diagram]

• The heaps are merged, but the result is not a leftist heap as 3 is unhappy.
• On the way back to the root and swap sub-heaps where necessary. Find the unhappy nodes – after updating the null path lengths.
- **Insert**: Inserting merges the existing heap with a heap of size one

**Delete Min**
6.7 The Skew Heap

- Skew heap – heap-ordered binary tree without a balancing condition.
- With these, there is no guarantee that the depth of the tree is logarithmic.
- It supports all operations in logarithmic **amortized time**.
- It is somewhat like a splay tree in the way the time bound is figured.
- It is similar to a leftist heap, but with less space.

**Merging**

- Many operations with heap-ordered trees can be done using merging.
- Operations:
  - Insert – create a one-node tree containing \( x \) and merge that tree into the priority queue.
  - Find minimum – return the item at the root of the priority queue.
  - Delete minimum – delete the root and merge its left and right subtrees.
  - Decrease the value of a node – assume that \( p \) points to the node in the priority queue. Lower the value of \( p \)'s key. Detach \( p \) from its parent, which yields two priority queues. Merge the two resulting priority queues.

**The Skew Heap – A Simple Modification**

- We can make a simple modification to the leftist heap and get similar results.
- We always merge with the right child, but after merging, we swap the left and right children for every node in the resulting right path of the temporary tree.
- Consider the example in Figure 3.

![Figure 3 Merging a skew heap](image)

- When a merge is performed in this way, the heap-ordered tree is also called a skew heap.
- Let’s consider this operation from a recursive point of view. Let \( L \) be the tree with the smaller root and \( R \) be the other tree.
  1. If one tree is empty, the other is the merged result.
  2. If \( t \) is the tree with the smaller value, Let \( t->right = merge(t->right, other) \)
  3. Swap the kids of \( t \)
- The result of child swapping is that the length of the right path will not be unduly large all the time.
- The amortized time needed to merge two skew heaps is \( O(\log n) \).
- Translated into code:
  ```cpp
  Node * SkewHeap::merge(Node * t1, Pair * t2)
  ```
{ Node *small, *big;
  if (t1==NULL) return t2;
  if (t2==NULL) return t1;
  if (t1->element< t2->element)
  {  small = t1; big = t2;
  }
  else
  {small = t2; big = t1;
  }
  small->right = merge(small->right, big);
  Node * temp = small->right;
  small->right = small->left;
  small->left = temp;
  return small;
}

For example:
6.8 Binomial Queues

- A binary heap provides $O(\log n)$ inserts and $O(\log n)$ deletes but suffers from $O(n \log n)$ merges
- A binomial queue offers $O(\log n)$ (average is constant time) inserts and $O(\log n)$ deletes and $O(\log n)$ merges
- A Binomial Queue is a collection of heap-ordered trees known as a forest. Each tree is a binomial tree. A recursive definition is:

  1. A binomial tree of height 0 is a one-node tree.
  2. A binomial tree, $B_k$, of height $k$ is formed by attaching a binomial tree $B_{k-1}$ to the root of another binomial tree $B_{k-1}$.

Examples

![Diagram of binomial trees]

Note that to store any number of nodes we need multiple $B_k$ trees.
The following pictures shows a binomial queue of 11 nodes

Questions:

1. How many nodes does the binomial tree $B_k$ have?
2. How many children does the root of $B_k$ have?
3. What types of binomial trees are the children of the root of $B_k$?
4. Is there a binomial queue with one node? With two nodes? With three nodes? … With \( n \) nodes for any positive integer \( n \)?

- When we want to find the minimum node, we just search all roots. How many roots could there be? What is the complexity?
- Consider Binary Addition

\[
\begin{array}{c}
1 & 0 & 1 & 1 & 0 & 1 & 0 \\
+ & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]

- Example: Merge two trees of the same size
Merging Binomial Forest

Queue 1

Queue 2

Merge same size trees starting at smallest

Queue 1

Queue 2

Priority Queues Page 12
Merge same size trees starting at smallest

Queue 1

Queue 2

Merge two B1 trees

Queue 1

Queue 2
There are three B2 trees.
Merge any two.

There are three B3 trees.
Merge any two.
Implementing Binomial Queues

Logical View

Physical View – instead of multiple child pointers, use a leftmost child and nextrightSibling

Link separate trees via sibling pointers
Questions:

- We now know how to merge two binomial queues. How do you perform an **insert**?
- How do you perform a **deleteMin**?
- What is the order of complexity of a **merge**? an **insert**? a **delete**?