Data Structures and Algorithms — CSci 230
Chapter 6 — Priority Queues

Overview

- We will cover up through section 6.7.
- The focus of the discussion will be the implementation of a priority queue as a *binary heap* or as a *leftist heap*.

Motivation

- Priority queues are used in prioritizing operations. Examples include jobs on a shop floor, packet routing in a network, scheduling in an operating system, or events in a simulation.
- Each item is stored in a priority queue using an associated "priority".
- The main operations are *insert* and *delete_min* (or *delete_max*, depending on the organization of the priority queue).
- *Binary heaps*, which can be implemented in an array and without pointers(!), allow these operations to be completed in $O(\log n)$ worst-case time, with constant average time for *insert*, and $O(n)$ time for construction.
    - **Exercise:** How do these times compare to those of ordinary queues, balanced binary search trees, and hash tables?
- *Leftist heaps* require $O(\log n)$ worst-case and average-case time for insert and delete. Two leftist heaps can be merged in $O(\log n)$ time, whereas merging two binary heaps requires $O(n)$ time.

Binary Heaps

- **Definition:** A binary heap is a complete binary tree such that at each internal node, p, the value stored is less than the value stored at either of p's children.
  - Binary heaps will be drawn as binary trees, but implemented in *arrays*!
  - Alternatively, the heap could be organized such that the value stored at each internal node is greater than the value at its children.
- The *delete_min* operation, which will be discussed in detail in class, depends heavily on the `Percolate_Down` function. This function is written here in terms of tree nodes with child pointers, but later it will be written in terms of array subscripts.

```
Percolate_Down( node * T )
{
  while ( T->Left != NULL ) {
    if ( T->Right != NULL
         && T->Right->Element < T->Left->Element )
      child = T->Right;
    else
      child = T->Left;

    if ( child->Element < T->Element ) {
      swap(child->Element, T->Element);
      T = child;
    }
    else
      break;
  }
}
```

- The *insert* operation, which will be discussed in detail in class, depends heavily on the `Percolate_Up` function. It assumes each node has a pointer to its parent.

```
Percolate_Up( node * T )
{
  while ( T->Parent != NULL ) {
    if ( T->Element < T->Parent->Element  ) {
      swap(T->Parent->Element, T->Element);
      T = T->Parent;
    }
    else
      break;
  }
}
```

- Both *delete_min* and *insert* are $O(\log n)$ in the worst-case, but *insert* is $O(1)$ in the average case.

Exercise

1. 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**!)

```
insert 5, insert 3, insert 8, insert 10, insert 1, insert 6,
delete_min,
insert 14, insert 2, insert 4, insert 7,
delete_min,
delete_min,
delete_min
```

Array implementation

- Number the nodes in the tree top to bottom and left to right, starting with 1. Place the values in an array, starting with subscript 1.
- For each subscript, i,
  - The parent, if it exists, is at location $\lfloor i/2 \rfloor$.
  - The left child, if it exists, is at location $2i$.
  - The right child, if it exists, is at location $2i + 1$.
- We will go over the author's code, in detail, in class. We will fix a number of problems and limitations in lab.
- The standard library (STL) `priority_queue` is implemented as a binary heap.

Exercises

1. What are some of the limitations of the author's code? What might you do differently? We will investigate these issues thoroughly in lab.

2. Suppose we used locations 0 through $n-1$ as the heap locations instead of 1 through n. For a value at location i, what are the array locations of its parent, its left child and its right child?

Build Heap

- Given an array with unsorted values in locations $1, \ldots, N$, the following code builds a heap (with `Percolate_Down` reimplemented to use array subscripts),

```
Build_Heap()
{
   for ( i=N/2; i>=1; i--)  Percolate_Down(i);
}
```

- We will prove in class that this requires $O(N)$ time.

Exercise

1. Consider the following list of values stored in locations 1 through 10 of an array.

   `10, 6, 12, 18, 5, 9, 7, 2, 4, 3`

   Show the contents of the array after `Build_Heap`.

Leftist Heaps

- 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.
- **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.
- **Theorem:** A leftist tree with $r > 0$ nodes on its right path has at least $2^r - 1$ nodes.
  - We will prove this in class 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.

Leftist Heap Operations

- The `insert` and `delete_min` operations will depend on the `merge` operation.
- Fundamental idea of merge: given two leftist heaps, with `h1` and `h2` pointers to their root nodes, and with `h1->Element <= h2->Element`, recursively merge `h1->Right` with `h2`, making the resulting heap `h1->Right`.
  - A simple trick is required to preserve the leftist property.
  - `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.
- We will examine the author's code for `merge` in class.

```
//  Here are the two functions used to implement leftist
//  heap merge operations.  Class "Left_Node" is a
//  normal tree node augmented with a member variable
//  "Npl" to record the minimum null path length from a
//  node.
//
//  Function Merge is the driver.  Function Merge1 does
//  most of the work.  These functions call each other
//  recursively.

template <class Etype>
Left_Node<Etype> *
Left_Heap<Etype>::
Merge( Left_Node<Etype> *H1, Left_Node<Etype> *H2 )
{
    if( H1 == NULL )
        return H2;
    if( H2 == NULL )
        return H1;
    if( H2->Element > H1->Element )
        return Merge1( H1, H2 );
    return( Merge1( H2, H1 ) );
}

template <class Etype>
Left_Node<Etype> *
Left_Heap<Etype>::
Merge1( Left_Node<Etype> *H1, Left_Node<Etype> *H2 )
{
    if( H1->Left == NULL ) // Single node.
        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;
}
```

Exercises

1. How can `merge` be used to implement `insert` and `delete_min`? Write pseudo-code to solve implement these.

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
```

Review Problems

1. Consider a binary heap, implemented, of course, as an array. Write a function to delete the element stored at location `i` of the heap. Assume that functions `Percolate_Down` and `Percolate_Up` exist, with prototypes

   ```
   void Percolate_Down( ElementType * heap, int size, int i);
   void Percolate_Up( ElementType * heap, int size, int i);
   ```

   where `heap` is the array of elements, and `size` is the current number of elements. Assume that `i` is correctly given to you, i.e., assume `1 <= i <= size`.

   Start from the following prototype

   ```
   void Delete( ElementType * heap, int & size, int i);
   ```

2. Consider a priority queue implemented as a heap. The heap is implemented using a vector, and it contains n values stored in subscript locations 1 through n. Assume it is a "max heap", so that the largest value is stored in subscript location 1.

   (a) What are the possible subscript locations for the third largest value in the heap (assuming all values are distinct)?

   (b) What is the range of possible subscript locations for the smallest value in the heap (again assuming all values are distinct)?

   (c) What is the worst-case time complexity required to find the minimum value in the heap?

3. Given an empty binary heap of integers, show the structure of the binary heap after the values 5, 2, 4, 6, 7, 1, 8 are inserted into it. (Note, you are not being asked to simulate the `Build_Heap` function.) Then show the heap after the minimum value is removed. You may show the heap as a tree rather than as an array. The heap is ordered so that the minimum value is at the root.

4. Suppose you are given a pointer, `Root`, to the root of a leftist heap of floating point values. Suppose you are also given a pointer, `P`, to a particular node in the leftist heap. Assume each `Leftist` node has the structure

```
struct Leftist {
  float Value;
  Leftist *Parent, *Left, *Right;
  int Npl;   // the null path length
};
```

You may assume the existence of a merge function with the following prototype:

```
Leftist* Merge( Leftist* t1_root, Leftist* t2_root )
```

which merges two leftist heaps, updates the null path lengths as necessary, and returns a pointer to the root of the resulting leftist tree.

(a) Write a function to remove the node pointed to by `P`. The function should be as efficient as possible. Here is the prototype:

```
void LeftistRemove( Leftist* & Root, Leftist* & P )
```

(b) What is the worst-case time required by the function? Assume N is the number of nodes in the entire tree. Explain briefly, but carefully.