10

Priority Queues

Just as people face waiting lines in many situations, the data being manipulated by a program often must spend some time "waiting in line" before they are processed. Programmers must devise and maintain the data structures in which data items wait for service. In Chapter 4, we saw two data types that give rules for waiting lines and some data structures that we could use to store data in a waiting line. Data items can be arranged according to far more intricate rules than these, however.

Recall for a moment our discussion in Section 4.1 of strategies for paying bills. We discussed two rules that the pile of unpaid bills could obey: oldest-first or FIFO, and newest-first or LIFO. Even as you read about them, it may have occurred to you that both of these strategies are flawed. For instance, if you pay bills in a strict FIFO order, you might renew a magazine subscription only to find that you no longer have enough money left for some large expense like tuition or housing. On the other hand, if you pay bills according to a strict LIFO rule, then a large bill on top of the stack could prevent you from paying older and smaller unpaid bills underneath it. In short, paying bills based on their time of arrival may be easy to understand, but it fails to do justice to common sense: we would like to pay bills in order of importance, so that we make sure to pay for vital goods and services before we pay for luxuries.

This chapter presents the priority queue data type that organizes data items in a waiting line according to a measure of their importance. The first three sections describe the basic data type and an elegant and practical data structure for priority queues. Next we shall
A heap ordered binary tree contains 10 items:

Figure 10.2

The data structure for priority queues is based on binary trees.

1. **Heap Insert** — insert an item with the priority given by key.
2. **Heap Remove** — remove an item with smallest priority.
3. **Heap Minimum** — return an item with the smallest priority.
4. **Heap Satisfy** — any heap violation on the priority queue.

The heap ordered binary tree at the right can store 10 items in O(log n) time. It allows our priority queue of items by priority.
1. **Heaps**

   a. **Heaps** are a data structure that are used to efficiently store and manage data in a way that allows for quick access to the maximum or minimum element. They are often used in priority queues or in situations where a sorted list is needed.

2. **Binary Heap**

   - A binary heap is a complete binary tree. It is a data structure that satisfies the heap property:
     - **Max Heap** (also called a **max-heap**): Each node's value is greater than or equal to its children's values.
     - **Min Heap** (also called a **min-heap**): Each node's value is less than or equal to its children's values.

3. **Heapsort**

   - Heapsort is a comparison-based sorting algorithm that uses a binary heap to sort elements.

4. **Heaps and Heapsort**

   - Heapsort works by first building a heap from the input data. Then, it repeatedly removes the maximum (or minimum) element from the heap and rebuilds the heap accordingly. This process is repeated until the heap is empty.

5. **Heaps and Priority Queues**

   - Heaps are often used in priority queues, where elements can be added and removed based on their priority.

---

**Figure 10.3**


---

**Figure 10.4**

Heap operations can be performed in O(\log n) time. Because the height of the tree is \( O(\log n) \), both of these operations take \( \log n \) time.

Insertion of a new item into the heap is performed by placing the item at the next available position in the tree and then percolating it up the tree until it is in the correct position. This is done by repeatedly moving the item up the tree if its key is larger than its parent's key.

Removal of the item with the smallest key is done by replacing the root with the last item in the heap and then percolating the new root down the tree until it is in the correct position. This is done by repeatedly moving the item down the tree if its key is smaller than its children's keys.

The figure on the left shows the result after 99 has been added to 99 at the root. The figure on the right shows the result after 19 has been added to 99 at the root. In both cases, the new item is added to the heap as described above.
PROGRAM 10.1 shows one version of implementation delete.\(\) This algorithm also deletes a value from the heap, and it does so in a way that preserves the heap property. The main idea is to maintain the invariant that the heap remains a valid priority queue after the operation is performed. This is achieved by first removing the root item from the heap and then performing a series of sift-down operations to restore the heap property.

The heap representation is shown in Figure 10.2. The heap is represented as a complete binary tree. The root node is the minimum element, and the tree is maintained in a way that ensures that the parent node is always less than or equal to its children. This property is preserved during the delete operation by performing sift-down operations as needed.

The delete operation consists of the following steps:

1. Remove the root node (the minimum element).
2. If the heap is not empty, the last element in the heap is moved to the root position.
3. Perform sift-down operations to maintain the heap property.

This process ensures that the heap remains a valid priority queue after the deletion, and it maintains the heap property as required.

For the implementation of this algorithm and the heap representation, see Figure 10.2. The implementation is shown in Program 10.1b, which is also shown in Program 10.1a. The heap is represented as a complete binary tree, and the root node is the minimum element. The heap property is maintained by performing sift-down operations as needed.
Huffman Trees

Function to delete the smallest item in a heap.

Program 10.16

break;

case 2:

default:

He successor (heap[heapsize] heap[heapsize - 1])

if (heap[heapsize] > heap[heapsize])

heap[heapsize] = heap[heapsize - 1];

break;

if (heap[heapsize] < heap[heapsize - 1])

heap[heapsize - 1] = heap[heapsize];

break;

default:

for (i = heapsize; i >= 1; i--)

if (heap[i] > heap[(i + 1) / 2])

swap(heap[i], heap[(i + 1) / 2]);

break;

continue;

break;

While (heap[heapsize] < heap[0])

heapsort(heap);

break;

default:

return heap[0];

break;


default deterrnination()()

Huffman Trees
Algorithm 10.1

1. **Tree-building step:**
   - The weight of each tree is the weight of the element it contains.
   - Each tree contains a single element.
   - Construct a forest of one-node trees.
   - Forest-building step:
     - If two trees contain the same set of items at their leaves, then the tree is a Huffman tree.
     - Huffman tree has the minimum weighted external path length.

2. **Huffman's algorithm to construct the tree of minimum weighted external path length:**

   - Start with a set of weighted items.
   - Merge the two items with the smallest weights to create a new node.
   - Remove the two items from the set.
   - Repeat until there is one tree.
   - The resulting tree is the Huffman tree.

Figure 10.11

The Huffman tree is the minimum weighted external path length.

The requirement that the encoded items reside at the leaves of a binary tree is important. To see why, consider the binary tree and its corresponding code. The ASII of RBCDAC corresponds to a complete binary tree of height 7. The code for the character 'a' is '10010'. Each character in the tree is represented by a sequence of characters. We can use this to encode a character. Therefore, a binary tree is necessary for the natural relationship between binary codes and binary trees.

A natural relationship between binary codes and binary trees.

Frequent symbols in short codes and infrequent ones in long codes.
Huffman tree construction

Function to read letters and associate weights, and construct a forest for

PROGRAM 10.2b

```java
private node node = readNode();
while node != null
{
    System.out.println(node.value);
    node = readNode();
}
```

Total getNodes()

```java
return result;
result = readNode()
result = readNode()
result = readNode()
result = readNode()
```

Program 10.2a

```java
exit(0)?
printTree(result)
getNodes()
```
Changing Priorities

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3. The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

10.5

The operations shown in Figure 10.4 are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.

Figure 10.12

Operations on priority queues. This section describes some other common operations shown in Figure 10.4 that are basic to the definition of the Fibonacci heap. We have implemented in Section 10.3.
In summary and perspective,

Property One uses

Combining Property One

Sometimes we must maintain a collection of property values to accommodate dependencies in the variable. As a result, we choose a sorted list data structure for a property. The operations are

\[
\begin{align*}
(1+\gamma)(1-x) & = (1-\gamma)x \\
((1+i) - \gamma Z & = 1 \gamma Z \\
(I+1)(1-x) & = (1-x)1 \gamma Z \\
(I+1) - \gamma Z & = 1 - \gamma Z \\
(I-1) - \gamma Z & = 1 - \gamma Z \\
\end{align*}
\]
Theorem 6.2.3 (Lemon, Karp, and Tarjan). A maximum heap is a structure that satisfies the following properties:

1. The root is the greatest element.
2. The children of each node are less than or equal to the parent node.
3. The heap is complete, meaning that if a node has a right child, it also has a left child.

The maximum heap can be constructed in O(n) time using a bottom-up approach. Here, we define a function `build_max_heap` that takes an array and constructs the maximum heap in place.

```
function build_max_heap(arr)
    n = length(arr)
    for i from n/2 downto 1
        max_heapify(arr, i)
    end for
end function
```

The `max_heapify` function ensures that the sub-tree rooted at `i` is a max heap.

```
function max_heapify(arr, i)
    l = left(i)
    r = right(i)
    largest = i
    if l <= n and arr[l] > arr[i]
        largest = l
    end if
    if r <= n and arr[r] > arr[largest]
        largest = r
    end if
    if largest != i
        swap(arr[i], arr[largest])
        max_heapify(arr, largest)
    end if
end function
```

The `swap` function simply swaps two elements in the array.

The proof of these properties involves a bottom-up approach that checks the heap properties for each node from the bottom up. The proof is by induction on the number of nodes below each level of the heap. The base case (level 1) is trivial, and the inductive step involves ensuring that the heap property is maintained when elements are added or removed from the heap.


REFERENCES

operation uses (O(n)) time.
insertion and deletion on a list of nodes so that each
39 nodes in the sorted list can be shown to be important (important)
and requires only O(n) time.

3. Use the preceding result to show how to merge two lists of nodes
ordered from the root by passing only through right edges.
The right path of a binary tree is defined by the nodes that can be
traversed from the root. As such, the left path of the root is no
longer than the height of the right path of a list of nodes with 1 nodes.

30. Draw the shortest and longest path of each node.

29 second as a local minima.

Lefthand leaves were inserted by:)

\text{The complete list of all leaves is:}
\text{\texttt{:\text{\texttt{L}}}}
\text{\texttt{R}}\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}
\text{\texttt{L}}\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}\text{\texttt{R}}
\text{\texttt{R}}\text{\texttt{L}}\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}
\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}
\text{\texttt{R}}\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}
\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}\text{\texttt{R}}
\text{\texttt{R}}\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}
\text{\texttt{L}}\text{\texttt{R}}\text{\texttt{L}}
\text{\texttt{R}}\text{\texttt{L}}
\text{\texttt{L}}
\text{\texttt{R}}

The above result on merging two binary trees means that we can

EXERCISES

Finite and infinite.

Sequence of 1's and 0's can be stored in a non-binary.

28 show that we start with one-node balanced binary.

27 Use Exercise 26 to show how to perform deletion on a balanced
binary search tree.

26 show what we derive from the use of a balanced tree. If the result is

25 shows what we mean by a balanced tree.

Two fundamental items in time, space, and the merging of

Property Ones

246