void pop();                // removes element from front
protected:
  vector<T> _;               // dynamic array for queue elements
  void heapifyDown();        // restores the heap property
  void heapifyUp();          // restores the heap property

};

This implementation is similar to the vector implementation of the Stack class template (Problem 5.13 on page 105). Most of the required member functions for the PriorityQueue class template call the corresponding member function for the vector class template.

template <class T>
PriorityQueue<T>::PriorityQueue()
{
}

template <class T>
PriorityQueue<T>::PriorityQueue(const PriorityQueue& q) : _{q._}
{
}

template <class T>
PriorityQueue<T>::~PriorityQueue()
{
}

template <class T>
PriorityQueue<T>& PriorityQueue<T>::operator=(const PriorityQueue& q)
{ _= q._;
}

int PriorityQueue<T>::size() const
{ return _.size();
}

bool PriorityQueue<T>::empty() const
{ return _.empty();
}

const T& PriorityQueue<T>::top() const
{ return _.front();
}

void PriorityQueue<T>::pop()
{ _.front() = _.back();       // delete the front element
  _.pop_back();               // move the back element to the front
  heapifyDown();              // restore the heap property
}

void PriorityQueue<T>::push(const T& x)
{ _.push_back(x);            // insert the new element at the back
  heapifyUp();               // restore the heap property
}

The heapifyDown() function percolates the root element down toward the leaf level in order to restore the heap property. It does that by swapping each parent along the root-to-leaf
path with its older child. Since the loop control variable \( i \) is doubled on each iteration, the entire process requires no more than \( \lg n \) iterations.

```cpp
template <class T>
void PriorityQueue<T>::heapifyDown()
{
    int n=__.size(), j;
    for (int i=0; i<n/2; i=j)
    { j=2*i+1; // _[j] and _[j+1] are the children of _[i]
        if (j<n && _[j]<_[j+1]) ++j;
        if (_[i] >= _[j]) break;
        swap(_[i],_[j]);
    }
}
```

The `heapifyUp()` function percolates the last leaf element up toward the root in order to restore the heap property. It does that by swapping each child along the root-to-leaf path with its younger parent. Since the loop control variable \( j \) is halved on each iteration, the entire process requires no more than \( \lg n \) iterations.

```cpp
template <class T>
void PriorityQueue<T>::heapifyUp()
{
    int n=__.size(), i;
    for (int j=n-1; j>0; j=i)
    { i=(j-1)/2; // _[i] is the parent of _[j]
        if (_[j] <= _[i]) break;
        swap(_[j],_[i]);
    }
}
```

The implementation of a priority queue as a heap data structure is very efficient. Both the `push()` and the `pop()` functions run in \( O(\lg n) \) time.

### 12.8 APPLICATIONS OF PRIORITY QUEUES

Like ordinary FIFO queues, priority queues are widely used in simulations. (See Section 6.3 on page 112.) The following example is typical.

**EXAMPLE 12.5 Simulating a Print Queue for a Local Area Network Pruner**

On a local area network ("LAN"), individual printers are used to service the print requests for all computers on the network. If a first-come-first-serve algorithm is used, then the printer can use an ordinary FIFO queue to hold the jobs that arrive while it is busy. But another common algorithm is to process the shorter jobs before the longer jobs among those that are waiting. This requires a priority queue, where the shorter jobs are given higher priority over the longer jobs.

The following program simulates this process.

```cpp
#include <iostream>  // defines the cout object
#include <queue>     // defines the priority_queue class template
#include "Random.h" // defines the Random class
using namespace std;

const int TICKS=60;  // 60 seconds
const float MTBA=4;  // mean time between arrivals
const float MNOP=5;  // mean number of pages per job
```
class Exp : public Random
{
    // generates random numbers that are exponentially distributed
    public:
        Exp(float mean) : _mean(mean) {}
        float next() { return -_mean * log(1.0 - real()); }
    private:
        float _mean;
};

class Job
{
    // instances represent print jobs:
    friend bool operator<(const Job& job1, const Job& job2)
    { // smaller jobs have higher priority:
        return (job1._pages > job2._pages);
    }
    friend ostream& operator<<(ostream& ostr, const Job& job)
    { ostr << "Job #" << job._id << "(" << job._pages << "pp)";
        return ostr;
    }
    static int _n;    // used to number the jobs: #0, #1, #2, etc.
    static Exp _gen;  // generates page counts
    public:
        Job(int time=0) : _id(_n++), _arrived(time),
            _pages(1 + int(_gen.next())) { }
        int id() { return _id; }
        void process() { --_pages; }  // simulates printing one page
        bool is_finished() { return (_pages == 0); }
    protected:
        int _id;       // identification number
        int _arrived;  // arrival time number
        int _pages;    // number of pages
    }

int Job::_n = 0;      // initializes Job counter
Exp Job::_gen(MNOP);  // defines page count generator

class Source
{
    // a Source object generates print jobs
    public:
        Source(float iat) : _gen(Exp(iat)), _arrival_time(0) { }
        Job* arrival(int t)
        { // returns pointer to new job if one is to arrive at time t:
            Job* job = 0;
            if (t == _arrival_time)
            { job = new Job(t);
                _arrival_time += int(_gen.next()) + 1;  // add random number
            }
            return job;
        }
    protected:
        Exp _gen;           // random number generator
        int _arrival_time;  // time of next print job arrival
    }
string size(priority_queue<Job>); // returns a string that symbolizes the size of the priority queue

int main()
{
    Source source(MTBA); // an object that generates print jobs
    priority_queue<Job> pq; // the priority queue for the print jobs
    Job* current_job=0; // the address 0 indicates no current job
    int cjid; // the current job’s identification number

    for (int t=0; t<TICKS; t++)
    {
        // each iteration represents one tick of the clock
        cout << "\n" << t << ": " << // print the current time cycle number
            if (new_job != 0)
                { pq.push(*new_job); // push it onto the queue
                    cout << *new_job << " arrived: " << // and report it
                } cout << size(pq); // report size of the queue
        if (current_job == 0 && !pq.empty()) // start next job from queue
        { Job job = pq.top(); // copy highest priority job in queue
            pq.pop(); // remove it from the queue
            current_job = &job; // save its address
            cjid = current_job->id(); // save its identification number
            cout << ", start #" << cjid << ": " << size(pq); // report
        } if (current_job != 0) // a current job exists
        { current_job->process(); // process the current job
            cout << ", process #" << cjid; // report action
            if (current_job->is_finished())
            { current_job = 0; // delete the current job
                cout << ", end #" << cjid; // report action
            } }
    }
    string size(priority_queue<Job> pq)
    { string s="[
    for (int i=0; i<pq.size(); i++)
        s += " *
    s += "]";
    return s;
    }

    The output is shown on the next page.
    The Exp class here is the same as that used in Example 6.2 on page 112.

    The Job class produces objects that represent print jobs that are sent to the LAN printer. The
    overloaded comparison operator < is required for objects stored in a priority_queue. Note that
    job1 has a lower priority than job2 if it has more pages. The overloaded output operator << is used to
    report information on the jobs as they arrive in the simulation. Each job has a unique numeric identification
    _id, its arrival time _arrived, and its current size _pages. Its identification number _id is given
    by the static field _n, and its initial size _pages is generated by the static object _gen. That object
    generates exponentially distributed random numbers with mean value MNOP, which is initialized at 5
    pages. Note in the run shown below that the first 10 jobs have 7, 3, 4, 3, 2, 2, 7, 11, 1, and 6 pages. The
The _pages field is initialized at 1 + int(_gen,.next()) to assure that no job starts with 0 pages. The process() function simulates the printing of one page, and the is_finished() function determines whether all of the pages have been printed.

Note that the two static fields Job::_n and Job::_gen must be defined and initialized outside of the Job class. The integer _n is initialized to 0 for the identification of the first job, and the object _gen is initialized by the Exp constructor, passing it the MNOP constant 5 pages.

The Source class is used to instantiate a single class that generates all the print jobs. It uses its _gen field to generate exponentially distributed random numbers for the times between job arrivals, with a mean value MTBA of 4 seconds between arrivals. Note in the run shown below that the first 10 jobs arrive at times 0, 6, 8, 11, 13, 15, 17, 23, 32, and 33 seconds; so the generated interarrival times were 6, 2, 3, 2, 2, 6, 9, and 1 seconds, actually averaging 33/9 = 3.67 seconds between arrivals. The expression int(_gen,.next())+1 is added to each arrival time to assure that no two jobs arrive at the same time. Note that the Source constructor uses the expression Exp(iat) to initialize its _gen object. This expression invokes the Exp constructor, passing it inter-arrival time iat. In the main() function, this value is the MTBA constant 4.

The Source::arrival() function is used to produce print jobs. Each time it is called, it uses the expression (t == _arrival_time) to determine whether it is time to generate a new print job. If it is, then the expression new Job(t) is used to construct a new print job, “timestamping” it with the current time t. Then it updates its _arrival_time field, setting it to the time for the next print job to arrive. The function returns a Job* pointer, which is either points to a new job or is 0 (i.e., null), according to whether it was time for a new job to be generated. The main() function calls the Source::arrival() function once for each iteration of its for loop. Note that in the run shown below, the arrival() function returned 0 except on iterations 0, 6, 8, 11, 13, 15, 17, 23, 32, 33, etc.

The main() program uses a for loop for the simulation. It iterates once for each “tick” of the clock. First it calls Source::arrival() to get the next print job if it has arrived, and pushes it onto the priority queue pq. Then it checks whether there is a job currently being printed; if not, and if the queue has any jobs waiting, then it removes the highest priority job (the one with the fewest pages) from the queue and makes it the current job by assigning its address to the current_job pointer. Finally, if a current job exists calls the process() function to simulate printing one page for that job, and then deletes it if it is finished.

### Review Questions

12.1 What are the two main applications of heaps?
12.2 What is the difference between a queue and a priority queue?
12.3 Why are heaps used to implement priority queues?

### Problems

12.1 Determine which of the following binary trees is a heap.

\[ a. \quad b. \quad c. \]

\[
\begin{array}{c}
\text{a.}
\begin{array}{c}
68 \\
33
\end{array}
\begin{array}{c}
33 \\
55
\end{array}
\begin{array}{c}
55 \\
77
\end{array}
\begin{array}{c}
88 \\
66
\end{array}
\end{array}
\quad
\begin{array}{c}
\text{b.}
\begin{array}{c}
88 \\
77
\end{array}
\begin{array}{c}
44 \\
33
\end{array}
\begin{array}{c}
66 \\
55
\end{array}
\end{array}
\quad
\begin{array}{c}
\text{c.}
\begin{array}{c}
55 \\
44
\end{array}
\begin{array}{c}
66 \\
88
\end{array}
\begin{array}{c}
33 \\
22
\end{array}
\end{array}
\end{array}
\]
12.2 Determine which of the following arrays have the heap property.

a. 0 1 2 3 4 5 6 7 88 66 44 33 55 77 32
b. 0 1 2 3 4 5 6 7 88 77 66 55 33 22 44

c. 0 1 2 3 4 5 6 7 88 44 77 22 33 55 66
d. 0 1 2 3 4 5 6 7 88 77 55 44 33 22 55

e. 0 1 2 3 4 5 6 7 88 66 77 22 33 44 55
f. 0 1 2 3 4 5 6 7 88 77 22 33 44 55 66

12.3 Show the heap after inserting each of these keys in this order: 44, 66, 33, 88, 77, 55, 22.

12.4 Show the array obtained from the natural map of each of the heaps obtained in Problem 12.3.

12.5 Prove that every subtree of a heap is a heap.

Answers to Review Questions

12.1 Heaps are used to implement priority queues and to implement the Heap Sort (see Section 13.8 on page 257).

12.2 Elements are removed from a queue in the same order in which they are inserted: first-in–first-out. Elements in a priority queue must have an ordinal key field which determines the priority order in which they are to be removed.

12.3 Heaps are used to implement priority queues because they allow $O(\log n)$ insertions and removals. This is because both the push() and the pop() functions are implemented by traversing a root-to-leaf path through the heap. Such paths are no longer than the height of the tree which is at most $\log n$.

Solutions to Problems

12.1 a. This is not a heap because the root-to-leaf path \{ 88, 44, 77 \} is not nonincreasing (44 < 77).

b. This is a heap.

c. This is not a heap because the root-to-leaf path \{ 55, 33, 44 \} is not nonincreasing (33 < 44) and the root-to-leaf path \{ 55, 77, 88 \} is not nonincreasing (55 < 77 < 88).

d. This is not a heap because the binary tree is not complete.

e. This is a heap.

f. This is not a heap because the tree is not binary.

12.2 a. This array does not have the heap property because the root-to-leaf path \{ a[1], a[3], a[6] \} = \{ 88, 44, 77 \} is not nonincreasing (44 < 77).

b. This array does have the heap property.

c. This array does have the heap property.

d. This array does not have the heap property because its data elements are not contiguous: it does not represent a complete binary tree.

e. This array does have the heap property.

f. This array does not have the heap property because the root-to-leaf path \{ a[1], a[3], a[6] \} = \{ 88, 22, 55 \} is not nonincreasing (22 < 55) and the root-to-leaf path \{ a[1], a[3], a[7] \} = \{ 88, 22, 66 \} is not nonincreasing (22 < 66).
12.3 Inserting the keys 44, 66, 33, 88, 77, 55, 22 into a heap:

12.4 The arrays for the heaps in Problem 12.3:

12.5 Theorem. Every subtree of a heap is also a heap.

Proof: Let $T$ be a heap, and let $S$ be a subtree of $T$. By definition, $T$ is a complete binary tree with the heap property. Therefore by the theorem of Problem 10.21 on page 221, $S$ is also a complete binary tree. Let $x$ be the root of $S$, and let $p$ be any root-to-leaf path in $S$. Then $x$ is an element of $T$ since $S$ is a subtree of $T$, and there is a unique path $q$ in $T$ from $x$ to the root of $T$. Also, $p$ is a path in $T$ that connects $x$ to a leaf of $T$ since $S$ is a subtree of $T$. Let $q^{-1}$ represent the reverse of the path $q$, and let $q^{-1}p$ represent the concatenation of $q^{-1}$ with $p$ in $T$. Then $q^{-1}p$ is a root-to-leaf path in $T$. Hence the elements along $q^{-1}p$ must be nonincreasing because $T$ has the heap property. Therefore the elements along $p$ are nonincreasing. Thus $S$ also has the heap property.

Q.E.D.