Priority Queues

A **queue** removes the oldest item from the list.

A **priority queue** removes the item of highest priority from the list.

- Each item has an associated priority level – highest priority is 0.
- When several items have the same priority level, these items are served in FIFO/FCFS order like in a queue.

Examples:

- applications (tasks) in an operating system (e.g., on Windows 98: Windows explorer task, printer task, Word task, etc.)

1

Priority Queue Implementation

```
#ifndef PRIORITYQUEUE_CLASS
#define PRIORITYQUEUE_CLASS

#include <iostream.h>
#include <stdlib.h>

// maximum size of the priority queue array
const int MaxPQSize = 50;

class PQueue
{
 . . .
}

#endif // PRIORITYQUEUE_CLASS
```

2

3

4

apqueue.h

```
class PQueue
{
   private:
      // priority queue array and count
      int count;
      T pqlist[MaxPQSize];

   public:
      // constructor
      PQueue (void);

      // priority queue modification operations
      void PQInsert(const T& item);
      T PQDelete(void);
      void ClearPQ(void);

      // priority queue test methods
      int PQEmpty(void) const;
      int PQFull(void) const;
      int PQLength(void) const;
};
```

5

```
// initialize priority queue count
PQueue::PQueue (void) : count(0)
{}

// return number of list elements
int PQueue::PQLength(void) const
{
   return count;
}

// test for an empty priority queue
int PQueue::PQEmpty(void) const
{
   return count == 0;
}

// test for a full priority queue
int PQueue::PQFull(void) const
{
   return count == MaxPQSize;
}

// clear the priority queue by resetting count to 0
void PQueue::ClearPQ(void)
{
   count = 0;
}
```

6

```
// insert item into the priority queue
void PQueue::PQInsert (const T& item)
{
   if (count == MaxPQSize) {
      cerr<< "Priority queue overflow!"<<endl;
      exit(1);
   }

   pqlist[count] = item;
   count++;
}
```



```
// delete an element from priority queue and return its value
T PQueue::PQDelete(void)
{
   T min;
   int i, minindex = 0;

   if (count > 0) {
      min = pqlist[0];

      for (i = 1; i < count; i++)
         if (pqlist[i] < min) {
            min = pqlist[i];
            minindex = i;
         }

      pqlist[minindex] = pqlist[count-1];
      count--;
   }
   else {
      cerr << "Priority queue is empty!!" << endl;
      exit(1);
   }
   return min;
}
```


Application: Company Support Services

A company has three types of employees:

- managers
- supervisors
- workers

Each employee can request services by filling out a form that includes

- category of person requesting service
- ID number for the task
- length of time the task will take.

Process forms that are stored in a file and print detail of each task and the total number of minutes spent servicing each category of employee in the company.

<Input file "job.dat"> : contains a list of job requests.

```
M 300 20
W 301 30
M 302 40
S 303 10
S 304 40
M 305 70
W 306 20
W 307 20
M 308 60
S 309 30
```

M : manager
S : supervisor
W : worker

Priority of
M < S < W

amount of time the job will take

ID number for the job

category of the employee requesting service

<Run of Program 5.6>

Category	Job ID	Job Time
Manager	300	20
Manager	302	40
Manager	308	60
Manager	305	70
Supervisor	309	30
Supervisor	303	10
Supervisor	304	40
Worker	306	20
Worker	307	20
Worker	301	30

Total Job Usage

Manager	190
Supervisor	80
Worker	70

job.h (1)

```
#ifndef JOBREQUEST
#define JOBREQUEST

// employee priority level (Manager=0, etc)
enum Staff {Manager, Supervisor, Worker};

// record defining a job request
struct JobRequest
{
   Staff staffPerson;
   int jobid;
   int jobTime;
};

// overload "<" to compare two JobRequests
int operator< (const JobRequest& a,
               const JobRequest& b)
{
   return a.staffPerson < b.staffPerson;
}
```

job.h (2)

```
// print a JobRequest record
void PrintJobInfo(JobRequest PR)
{
   switch (PR.staffPerson) {
      case Manager:      cout << "Manager       ";
                         break;
      case Supervisor:   cout << "Supervisor    ";
                         break;
      case Worker:       cout << "Worker        ";
                         break;
   }
   cout << PR.jobid << "       " << PR.jobTime << endl;
}
```


job.h (3)

```
#include <iomanip.h>     // use manipulator setw

// print total job time allocated to each employee category
void PrintJobSummary(int jobServicesUse[])
{
   cout << "\nTotal Support Usage\n";
   cout << "   Manager    " << setw(3)
        << jobServicesUse[0]  << endl;
   cout << "   Supervisor " << setw(3)
        << jobServicesUse[1]  << endl;
   cout << "   Worker     " << setw(3)
        << jobServicesUse[2]  << endl;
}

#endif  // JOBREQUEST
```


job.cpp (1)

```
#include <iostream.h>
#include <fstream.h>
#pragma hdrstop

#include "job.h"

// priority queue elements are of type JobRequest
typedef  JobRequest T;

#include "apqueue.h"      // include the PQueue class
```


job.cpp (2)

```
void main()
{
   // handle up to 25 job requests
   PQueue jobPool;

   // job requests are read from fin
   ifstream fin;

   // time spent working for each category of employee
   int jobServicesUse[3] = {0, 0, 0};

   JobRequest PR;
   char ch;

   // open "job.dat" for input. exit program if open fails
   fin.open("job.dat", ios::in | ios::nocreate);
   if (!fin) {
      cerr << "Cannot open file 'job.dat'" << endl;
      exit(1);
   }
   . . .
```

job.cpp (3)

```
. . .
// read file. insert each job into priority queue jobPool.
while (fin >> ch) {
    // assign staffPerson field
    switch(ch) {
        case 'M':  PR.staffPerson = Manager;
                   break;
        case 'S':  PR.staffPerson = Supervisor;
                   break;
        case 'W':  PR.staffPerson = Worker;
                   break;
        default:   break;
    }
    // read the integer jobid and jobTime fields of PR
    fin >> PR.jobid;
    fin >> PR.jobTime;

    // insert PR into the priority queue
    jobPool.PQInsert(PR);
}
. . .
```

19

job.cpp (4)

```
    . . .
// delete jobs from priority queue and print information
    cout << "Category    Job ID    Job Time\n\n";
    while (!jobPool.PQEmpty()) {
        PR = jobPool.PQDelete();
        PrintJobInfo(PR);
        // accumulate job time for each category of employee
        jobServicesUse[int(PR.staffPerson)]
                += PR.jobTime;
    }

    PrintJobSummary(jobServicesUse);
}
```

20

node.h

```
#ifndef NODE_CLASS
#define NODE_CLASS

template <class T>
class Node
{
  private:
     Node<T> *next;
  public:
     T data;

    // constructor
    Node(const T& item, Node<T>* ptrnext=NULL);

    // list modification methods
    void InsertAfter(Node<T> *p);
    Node<T> *DeleteAfter(void);

    // obtain the address of the next node
    Node<T> *NextNode(void) const;
};
. . .
#endif    // NODE_CLASS
```

21

```
// constructor. initialize data and pointer members
template <class T>
Node<T>::Node(const T& item, Node<T>*
  ptrnext)
      : data(item), next(ptrnext)
{ }

// return value of private member next
template <class T>
Node<T> *Node<T>::NextNode(void) const
{
   return next;
}

// insert a node p after the current one
template <class T>
void Node<T>::InsertAfter(Node<T> *p)
{
   // p points to successor of the current node,
   // and current node points to p.
   p->next = next;
   next = p;
}
```

22

```
// delete the node following current and return its
   address
template <class T>
Node<T> *Node<T>::DeleteAfter(void)
{
   // save address of node to be deleted
   Node<T> *tempPtr = next;

   // if there isn't a successor, return NULL
   if (next == NULL)
      return NULL;

   // current node points to successor of tempPtr.
   next = tempPtr->next;

   // return the pointer to the unlinked node
   return tempPtr;
}
```

23