ASS2

1. Introduction. This is a literate program which solves the problem set in Assignment two—queueing. The specification for this assignment states:

Your task for this assignment is to investigate some of the properties of queues and their management.

You should write a program which simulates the queuing and service of airline passengers.

Your program should first read in a file name from standard input and then open this file and read data from the named file.

The input file will contain the following data:

- *The number of first/business class servers in the system*
- *The number of tourist class servers in the system*
- *A set of passengers, each consisting of an arrival time, a class and a service time.*
  - *Class is an integer where* 0 *indicates a tourist class passenger and* 1 *indicates a first/business class passenger.*
- *This set is terminated by a dummy record with arrival time and service times all equal to* 0.
- *Note: the arrival times are sorted in ascending order.*

The simulation is to be of an airline check in system with two sets of servers, first/business class and tourist class, with a single queue associated with each set. Customers arrive in the system and are served by a server of the appropriate class. If all servers of a particular type are busy, the passenger will enter either the first/business or tourist class queue as appropriate.

The simulation should be run until the last passenger has left the system.

The simulation will run twice, in the first run each server will only serve passengers of the appropriate class. In the second run the first/business class servers will be able to serve passengers in the tourist class queue if no first/business class passengers are waiting.

Output, to standard output, will consist of the following data for each run:

- *Number of people served.*
- *Time last service request is completed.*
- *Average total service time (this includes time spent in a queue).*
- *Average total time in queue(s). Both overall and separate.*
- *Average length of queue. For each queue and overall.*
- *Maximum Length of queue. For each queue and overall.*
- *Total idle time for each server.*

2. We will start with the outline of the program.

using namespace std;
⟨ Headers 7 ⟩
⟨ Global data 3 ⟩
⟨ Prototypes for functions 41 ⟩
⟨ The main program 5 ⟩
⟨ Implementation of functions 42 ⟩

3. Data structures. For this program we will start by declaring the main global data structures that will be needed.

We will need five arrays to manage the simulation: these consist of heap, (unimaginatively called *heap*), used to manage the time sequence of events; two queues, *bQueue* and *tQueue* for each of business and tourist class passengers who have arrived but who cannot yet be served and two stacks, *bStack* and *tStack* for each holding idle business and tourist class servers.

We will create three of these arrays dynamically in the main program but we will make the heap globally available. Because I do not feel inclined to mess about with dynamic queue resizing, I will declare the queues as fixed size arrays of sufficient size to be safe.

We also should declare indices to keep track of the head and tail of each queue.

Although we need to maintain different information in each of the arrays, I will use a single aggregate struct, called *event*, in all of them. As I am not sure exactly what this will look like I will defer decision about its detail until later. At this point, the only thing I am sure of is that the *event* will need to hold a *time* field.

The heap, along with its size should be the only global variables needed.

```
⟨ Global data 3 ⟩ ≡
  struct event {
    float time;
    ⟨ Contents of the event struct 11 ⟩
  };
  event *heap;
  int heapSize;
```

This code is used in section 2.

4. The remaining arrays will only be accessed from *main* so we can make them local. We will declare a few extra, queue related, variables to make tracking statistics easier.

```
⟨ Variables of main 4 ⟩ ≡
  const int QUEUE_MAX = 500;
  event bQueue[QUEUE_MAX];
  int bHead;
  int bTail;
  int bqSize;
  int bqMax;
  float bqTotal;
  event tQueue[QUEUE_MAX];
  int tHead;
  int tTail;
  int tqSize;
  int tqMax;
  float tqTotal;
  int qMax;
  event *bStack;
  event *tStack;
```

See also sections 6, 10, 12, 14, 16, and 27.

This code is used in section 5.

5. Main. Ok, let's start writing *main*. The skeleton of the *main* program is as follows. The variable *pass* will be used to keep track of which run of the simulation we are conducting. We will close the input file at the end of the first, ready to be opened and read again in the second pass.

```
⟨ The main program 5 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 4 ⟩
    for (int pass = 1; pass ≤ 2; pass ++) {
      ⟨ Open and validate the input file 8 ⟩
      ⟨ Initialize the simulation 9 ⟩
      ⟨ Do the main loop 17 ⟩
      ⟨ Write the results 35 ⟩
      fin.close( );
    }
    ⟨ Finish up 39 ⟩
    return 0;
  }
```

This code is used in section 2.

6. The first thing we need to do is declare the variables we need to input words. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 4 ⟩ +≡
  char filename[20];
  ifstream fin;
```

7. We need a couple of header files *iostream* for stream-based input and *fstream* for managing files.

```
⟨ Headers 7 ⟩ ≡
#include <iostream>
#include <fstream>
```

See also section 38.

This code is used in section 2.

8. Now we can get the file opened and ready for input. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

Note that we only need to prompt for the filename the first time round.

```
⟨ Open and validate the input file 8 ⟩ ≡
  if (pass ≡ 1) {
    cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
    cin ≫ filename;
  }
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

This code is used in section 5.

9. We are now ready to start the program proper. For each pass we will follow the same process with only a small number of differences needed between the two runs. We will start with the initialization mechanism. Here we need to read in the number of servers of each class, create the arrays we need (on the first pass) and set them to suitable initial values.

We will start by setting up the servers. The variables *bCount* and *tCount* are the number of business and tourist class servers in the simulation. Initially,all servers are idle so we set the corresponding idle counters *bIdle* and *tIdle* appropriately. We also need to set the cumulative *busyDuration* to 0.0 for each server.

```
⟨ Initialize the simulation 9 ⟩ ≡
  fin ≫ bCount ≫ tCount;
  if (pass ≡ 1) {
     heap = new event[bCount + tCount + 1];
     bStack = new event[bCount];
     tStack = new event[tCount];
  }
  bIdle = bCount;
  tIdle = tCount;
  for (int i = 0; i < bCount; i++) bStack[i].busyDuration = 0.0;
  for (int i = 0; i < tCount; i++) tStack[i].busyDuration = 0.0;
```

See also sections 13 and 28.

This code is used in section 5.

10. Ok, we introduced a few new variables there.

```
⟨ Variables of main 4 ⟩ +≡
  int bCount, tCount, bIdle, tIdle;
```

11. We also introduced a new field for the **event** struct. The *busyDuration* field will record the total duration each individual server is busy.

```
⟨ Contents of the event struct 11 ⟩ ≡
  float busyDuration;
```

See also section 15.

This code is used in section 3.

12. We also zero out the queue indices as both queues are initially empty.

```
⟨ Variables of main 4 ⟩ +≡
  bHead = 0;
  bTail = 0;
  bqSize = 0;
  bqMax = 0;
  tHead = 0;
  tTail = 0;
  tqSize = 0;
  tqMax = 0;
  qMax = 0;
```

13. Our simulation starts at a time of 0.0 so we need to set the *now*.

We also need to get the first arrival into the heap. As well as recording the arrival time, *inArrival*, the class, *inClass* and the service duration, *inService* we set the duration of this passenger in the queue to 0.0. **Note:** The words time and duration as used in this document have distinctly different meanings. The word *time* is reserved for describing the instant when something happens; the word *duration* is reserved for describing how long something takes.

⟨ Initialize the simulation 9 ⟩ +≡
```
now = 0.0;
heapSize = 1;
heap[0].eventType = 0;
fin ≫ inArrival ≫ inClass ≫ inService;
heap[0].time = inArrival;
heap[0].passengerClass = inClass;
heap[0].serviceDuration = inService;
heap[0].queueDuration = 0.0;
```

14. We need to set up the clock. Rather than changing by a steady tick, the clock variable *now* will advance in variable sized chunks. This process is governed by the time of the **event** in heap[0].

⟨ Variables of main 4 ⟩ +≡
```
float now;
```

15. Our **event** has a few additional fields now. We identify what is happening using *eventType*, with 0 indicating an arrival, 1 indicating a tourist class server completion and 2 indicating a business class server completion. The *passengerClass* and *serviceDuration* components hold the data read from the input file. The *queueDuration* file will record the time, if any, spent by the passenger in a queue.

⟨ Contents of the **event** struct 11 ⟩ +≡
```
int eventType;
int passengerClass;
float serviceDuration;
float queueDuration;
```

16. we also need some extra variables to hold the input values from the file.

⟨ Variables of main 4 ⟩ +≡
```
float inArrival;
int inClass;
float inService;
```

17. The Main Loop. This is where all the interesting stuff happens. The event heap will be maintained as a min heap, ordered on the *time* field. Thus, whatever event is in *heap*[0] will be the next to occur.

The *eventType* of the next event will be used to distinguish between arrivals, *eventType* $\equiv 0$ and service completions, *eventType* $\neq 0$. Specifically, *eventType* $\equiv 1$ indicates a tourist class server and *eventType* $\equiv 2$ indicates a business class server completing a service respectively.

In either case, we advance the *now* to the time in *heap*[0] and proceed to handle the event.

The first thing we do, however, is to accumulate the queue size statistics. We do this by mutiplying the current queue sizes by the time that has elapsed since the last event.

```
⟨ Do the main loop 17 ⟩ ≡
  while (heapSize ≠ 0) {
    bqTotal += (heap[0].time − now) ∗ bqSize;
    tqTotal += (heap[0].time − now) ∗ tqSize;
    now = heap[0].time;
    if (heap[0].eventType ≡ 0) {
      ⟨ Process an arrival 18 ⟩
    }
    else {
      ⟨ Process a service completion 25 ⟩
    }
  }
```

This code is used in section 5.

18. Arrivals. When the event in *heap*[0] is an arrival we need to either add a new service completion to the heap or add the newly arrived passenger to the queue. We also need to get the next arrival, if there is one, onto the heap in place of the passenger who just arrived.

```
⟨ Process an arrival 18 ⟩ ≡
  ⟨ Serve or queue the current passenger 19 ⟩
  ⟨ Get the next passenger 24 ⟩
```

This code is used in section 17.

19. The first thing we need to do is to determine the class of the current passenger and whether there is a server available. We can then either create a service event, using an idle server, or add the arrival to the appropriate queue.

```
⟨ Serve or queue the current passenger 19 ⟩ ≡
  if (heap[0].passengerClass ≡ 0)      /* tourist arrival */
  {
    if (tIdle ≠ 0) {
      ⟨ Serve using a tourist class server 20 ⟩
    }
    else if (pass ≡ 2 ∧ bIdle ≠ 0) {
      ⟨ Serve using a business class server 21 ⟩
    }
    else {
      ⟨ Add a tourist class passenger to the queue 22 ⟩
    }
  }
  else {
    if (bIdle ≠ 0) {
      ⟨ Serve using a business class server 21 ⟩
    }
    else {
      ⟨ Add a business class passenger to the queue 23 ⟩
    }
  }
```

This code is used in section 18.

20. Tourist class servers will only ever serve tourist classs passengers. To add a new service event we combine the data for the current passenger from *heap*[0] and the data for an idle server from the top of *tStack*.

```
⟨ Serve using a tourist class server 20 ⟩ ≡
  heap[heapSize].time = now + heap[0].serviceDuration;
  heap[heapSize].eventType = 1;
  heap[heapSize].passengerClass = heap[0].passengerClass;
  heap[heapSize].serviceDuration = heap[0].serviceDuration;
  heap[heapSize].queueDuration = 0;
  tIdle−−;
  heap[heapSize].busyDuration = tStack[tIdle].busyDuration;
  siftUp(heapSize);
  heapSize++;
```

This code is used in section 19.

21. Business class servers usually serve business class passengers but can sometimes (in *pass* 2) serve tourist class passengers. The server data this time is taken from the top of *bStack*.

⟨ Serve using a business class server 21 ⟩ ≡

```
heap[heapSize].time = now + heap[0].serviceDuration;
heap[heapSize].eventType = 2;
heap[heapSize].passengerClass = heap[0].passengerClass;
heap[heapSize].serviceDuration = heap[0].serviceDuration;
heap[heapSize].queueDuration = 0;
bIdle−−;
heap[heapSize].busyDuration = bStack[bIdle].busyDuration;
siftUp(heapSize);
heapSize++;
```

This code is used in section 19.

22. If nobody can serve them, the tourist is moved from the heap to the tourist queue. The only information we need to maintian in the queue is the time the passenger started queueing and the service duration for this passenger.

⟨ Add a tourist class passenger to the queue 22 ⟩ ≡

```
if (tTail ≡ QUEUE_MAX) tTail = 0;
tQueue[tTail].time = now;
tQueue[tTail].serviceDuration = heap[0].serviceDuration;
tTail++;
tqSize++;
tqMax = max(tqMax, tqSize);
qMax = max(qMax, tqSize + bqSize);
```

This code is used in section 19.

23. Similarly, if nobody can serve them, the business class passenger is added to the business queue.

⟨ Add a business class passenger to the queue 23 ⟩ ≡

```
if (bTail ≡ QUEUE_MAX) bTail = 0;
bQueue[bTail].time = now;
bQueue[bTail].serviceDuration = heap[0].serviceDuration;
bTail++;
bqSize++;
bqMax = max(bqMax, bqSize);
qMax = max(qMax, tqSize + bqSize);
```

This code is used in section 19.

24. All that remains is to fix the arrival event. We read in the next arrival from the input file and, if it is valid, we replace the arrival on the top of the heap with the new one. otherwise we have just completed processing the last arrival and, in this case, we remove the arrival from the heap completely. In either case, we finish by restoring the heap.

```
⟨ Get the next passenger 24 ⟩ ≡
  $fin \gg inArrival \gg inClass \gg inService$;
  if ($inArrival > 0.0 \lor inService > 0.0$) {
    $heap[0].time = inArrival$;
    $heap[0].passengerClass = inClass$;
    $heap[0].serviceDuration = inService$;
    $heap[0].queueDuration = 0.0$;
  }
  else {
    $heapSize$−−;
    $heap[0] = heap[heapSize]$;
  }
  $siftDown(0)$;
```

This code is used in section 18.

25. Departures. Now we need to consider what happens when the next event is a service completion. We need to do two things here. First we accumulate the statistics for the departing passenger and then we fix the service completion event either keeping the server busy (if there are passengers waiting) or making them idle.

```
⟨ Process a service completion 25 ⟩ ≡
  ⟨ Update the passenger statistics 26 ⟩
  ⟨ Fix the server 29 ⟩
```

This code is used in section 17.

26. When a passenger ends service we need to add them into the service statistics. We will collect these statistics into arrays of length 3 where entry 0 is for tourist passengers, entry 1 for business class and entry 2 for the overall values.

We can also add the *serviceDuration* for this passenger to the cumulative server *busyDuration*.

```
⟨ Update the passenger statistics 26 ⟩ ≡
  if (heap[0].passengerClass ≡ 0) {
    totalService[0] += heap[0].serviceDuration + heap[0].queueDuration;
    pureService[0] += heap[0].serviceDuration;
    nServed[0]++;
  }
  else {
    totalService[1] += heap[0].serviceDuration + heap[0].queueDuration;
    pureService[1] += heap[0].serviceDuration;
    nServed[1]++;
  }
  totalService[2] += heap[0].serviceDuration + heap[0].queueDuration;
  pureService[2] += heap[0].serviceDuration;
  nServed[2]++;
  heap[0].busyDuration += heap[0].serviceDuration;
```

This code is used in section 25.

27. Ok, we need to declare and initialize these cumulative statistics arrays.

```
⟨ Variables of main 4 ⟩ +≡
  float totalService[3];
  float pureService[3];
  int nServed[3];
```

28. These arrays should be all set to zero at the start of each pass.

```
⟨ Initialize the simulation 9 ⟩ +≡
  for (int i = 0; i < 3; i++) {
    totalService[i] = 0.0;
    pureService[i] = 0.0;
    nServed[i] = 0;
  }
```

29. The passenger has now left the building. We now consider the server.

```
⟨ Fix the server 29 ⟩ ≡
  if (heap[0].eventType ≡ 1) {
    ⟨ Fix a tourist class server 30 ⟩
  }
  else {
    ⟨ Fix a business class server 31 ⟩
  }
  siftDown(0);
```

This code is used in section 25.

30. Tourist class servers are the easiest to deal with. If there are any waiting tourist class passengers we keep the server busy by allocating a passenger from the tourist queue. If no passengers of the appropriate class are queued, the server becomes idle.

```
⟨ Fix a tourist class server 30 ⟩ ≡
  if (tHead ≠ tTail) {
    ⟨ Keep the server busy with a tourist class passenger 32 ⟩
  }
  else {
    ⟨ Make the server idle 34 ⟩
  }
```

This code is used in section 29.

31. The business class server is a little more complex (at least in the second pass). If there are any waiting business class passengers we keep the server busy by allocating a passenger from the business queue. In *pass* 2 we also check and, where available, allocate passengers from the tourist queue. If no passengers of the appropriate class are queued, the server becomes idle.

```
⟨ Fix a business class server 31 ⟩ ≡
  if (bHead ≠ bTail) {
    ⟨ Keep the server busy with a business class passenger 33 ⟩
  }
  else if (pass ≡ 2 ∧ tHead ≠ tTail) {
    ⟨ Keep the server busy with a tourist class passenger 32 ⟩
  }
  else {
    ⟨ Make the server idle 34 ⟩
  }
```

This code is used in section 29.

32. If the server is now going to serve a tourist class passenger we set the new service completion time to the current time, *now*, plus the *serviceDuration* for the dequeued passenger. When a passenger is removed from the queue we can calculate their *queueDuration* as the difference between the current time and their arrival time, stored in the *time* field of the queue entry. Finally, we update the queue, removing the passenger and decreasing its size.

```
⟨Keep the server busy with a tourist class passenger 32⟩ ≡
  heap[0].time = now + tQueue[tHead].serviceDuration;
  heap[0].passengerClass = 0;
  heap[0].queueDuration = now − tQueue[tHead].time;
  heap[0].serviceDuration = tQueue[tHead].serviceDuration;
  tHead++;
  if (tHead ≡ QUEUE_MAX) tHead = 0;
  tqSize−−;
```

This code is used in sections 30 and 31.

33. The process of serving a business class passenger is identical, the only difference being the the passenger is taken from the business queue.

```
⟨Keep the server busy with a business class passenger 33⟩ ≡
  heap[0].time = now + bQueue[bHead].serviceDuration;
  heap[0].passengerClass = 1;
  heap[0].queueDuration = now − bQueue[bHead].time;
  heap[0].serviceDuration = bQueue[bHead].serviceDuration;
  bHead++;
  if (bHead ≡ QUEUE_MAX) bHead = 0;
  bqSize−−;
```

This code is used in section 31.

34. If a server becomes idle they are moved back onto the appropriate idle stack and the heap shrinks by one. The cumulative *busyDuration* is the only information about the server that we need to maintain.

```
⟨Make the server idle 34⟩ ≡
  if (heap[0].eventType ≡ 1) {
    tStack[tIdle].busyDuration = heap[0].busyDuration;
    tIdle++;
  }
  else {
    bStack[bIdle].busyDuration = heap[0].busyDuration;
    bIdle++;
  }
  heapSize−−;
  heap[0] = heap[heapSize];
  siftDown(0);
```

This code is used in sections 30 and 31.

35. Output. At this point the simulation has completed a pass and we need to output the relevant statistics.

```
⟨ Write the results 35 ⟩ ≡
  cout ≪ fixed ≪ setprecision(2);
  cout ≪ "Pass␣" ≪ pass ≪ ":␣";
  if (pass ≡ 1) cout ≪ "Business␣servers␣exclusively␣serve␣business␣class" ≪ endl;
  else cout ≪ "Idle␣business␣servers␣may␣serve␣tourist␣class" ≪ endl;
  cout ≪ setw(50) ≪ "␣␣Time␣last␣service␣is␣completed:" ≪ setw(10) ≪ now ≪ endl;
  ⟨ Output the passenger statistics 36 ⟩
  ⟨ Output the server statistics 37 ⟩
```

This code is used in section 5.

36. Passenger stats.

```
⟨ Output the passenger statistics 36 ⟩ ≡
  cout ≪ "␣␣Business␣class␣passengers:" ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Number␣of␣people␣served:" ≪ setw(10) ≪ nServed[1] ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣total␣service␣time:" ≪ setw(10) ≪ totalService[1]/nServed[1] ≪
      endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣total␣time␣in␣service:" ≪ setw(10) ≪
      pureService[1]/nServed[1] ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣length␣of␣queue:" ≪ setw(10) ≪ bqTotal/now ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Maximum␣number␣queued:" ≪ setw(10) ≪ bqMax ≪ endl;
  cout ≪ "␣␣Tourist␣class␣passengers:" ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Number␣of␣people␣served:" ≪ setw(10) ≪ nServed[0] ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣total␣service␣time:" ≪ setw(10) ≪ totalService[0]/nServed[0] ≪
      endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣total␣time␣in␣service:" ≪ setw(10) ≪
      pureService[0]/nServed[0] ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣length␣of␣queue:" ≪ setw(10) ≪ tqTotal/now ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Maximum␣number␣queued:" ≪ setw(10) ≪ tqMax ≪ endl;
  cout ≪ "␣␣All␣passengers:" ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Number␣of␣people␣served:" ≪ setw(10) ≪ nServed[2] ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣total␣service␣time:" ≪ setw(10) ≪ totalService[2]/nServed[2] ≪
      endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣total␣time␣in␣service:" ≪ setw(10) ≪
      pureService[2]/nServed[2] ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Average␣length␣of␣queue:" ≪ setw(10) ≪ (bqTotal + tqTotal)/now ≪ endl;
  cout ≪ setw(50) ≪ "␣␣␣␣Maximum␣number␣queued:" ≪ setw(10) ≪ qMax ≪ endl;
```

This code is used in section 35.

37. Server stats. When the simulation is finished all the servers are idle so we can get the information we need from the stack.

```
⟨ Output the server statistics 37 ⟩ ≡
  cout ≪ "␣␣Business␣class␣servers:" ≪ endl;
  for (int i = 0; i < bCount; i++) cout ≪ "␣␣␣␣Total␣idle␣time␣for␣business␣class␣server␣" ≪
        setw(2) ≪ i + 1 ≪ ":␣" ≪ setw(10) ≪ now − bStack[i].busyDuration ≪ endl;
  cout ≪ "␣␣Tourist␣class␣servers:" ≪ endl;
  for (int i = 0; i < tCount; i++) cout ≪ "␣␣␣␣Total␣idle␣time␣for␣tourist␣class␣server␣" ≪
        setw(2) ≪ i + 1 ≪ ":␣␣" ≪ setw(10) ≪ now − tStack[i].busyDuration ≪ endl;
  cout ≪ endl;
```

This code is used in section 35.

38. to use *fixed*, *setw* and *setprecision* we need another header file.

⟨ Headers 7 ⟩ +≡
#include `<iomanip>`

39. All Done. All that remains is to tidy up the dynamic memory.

```
⟨ Finish up 39 ⟩ ≡
  delete[ ] heap;
  delete[ ] bStack;
  delete[ ] tStack;
```

This code is used in section 5.

40. Functions. We only have two functions to implement—the ones that maintain the heap.

41. First the prototypes:

```
⟨ Prototypes for functions 41 ⟩ ≡
  void siftUp(int);
  void siftDown(int);
```

This code is used in section 2.

42. and now the implementations. First *siftUp*( ). We recusively compare the current entry, *heap*[*i*] with its parent. *heap*[*p*] swaping the entries if the parent has a later *time* than the child.

```
⟨ Implementation of functions 42 ⟩ ≡
  void siftUp(int i)
  {
    static event temp;
    if (i ≡ 0) return;
    int p = (i − 1)/2;
    if (heap[p].time ≤ heap[i].time) return;
    temp = heap[p];
    heap[p] = heap[i];
    heap[i] = temp;
    siftUp(p);
    return;
  }
```

See also section 43.

This code is used in section 2.

43. Finally we implement *siftDown*( ). This time we compare the current entry, *heap*[*i*], with the child, *heap*[*c*], having the smaller *time*, again swapping and recursing as needed. This function is a little messier as we have somewhere between 0 and 2 children.

```
⟨ Implementation of functions 42 ⟩ +≡
  void siftDown(int i)
  {
    static event temp;
    int c = 2 ∗ i + 1;
    if (c ≥ heapSize) return;
    if (c + 1 < heapSize ∧ (heap[c + 1].time < heap[c].time)) c++;
    if (heap[i].time ≤ heap[c].time) return;
    temp = heap[c];
    heap[c] = heap[i];
    heap[i] = temp;
    siftDown(c);
    return;
  }
```

44. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.