```
--file Sort.mesa
--last modified by Bruce on  July 30, 1978  4:30 PM
-- translated from Ed McCreight's BCPL by Jim Frandeen

DIRECTORY
  AltoDefs: FROM "AltoDefs",
  FSPDefs:  FROM "fspdefs" USING [NoRoomInZone, NodeOverhead],
  GPsortDefs:  FROM "GPsortDefs",
  InlineDefs:  FROM "inlinedefs" USING [COPY],
  StreamDefs:  FROM "StreamDefs";

DEFINITIONS FROM GPsortDefs;

Sort: PROGRAM
  IMPORTS FSPDefs, GPsortDefs, StreamDefs
  EXPORTS GPsortDefs =
BEGIN

NFiles: CARDINAL = 3;  -- number of scratch files

bufferSize: INTEGER;
compareProc: CompareProcType;
files: ARRAY [0..NFiles] OF FdHandle;
firstFreeEnt: CARDINAL;  -- 1 + end of unsorted part of heap vector
getProc: GetProcType;
heap: DESCRIPTOR FOR ARRAY OF ItemHeaderHandle;
heapSize: CARDINAL;  -- end of heap-sorted part of heap vector
inputFinished: BOOLEAN;
itemIsLeftOver: BOOLEAN;
leftoverItem: ItemHandle;
leftoverItemLen: ItemLength;
level: CARDINAL;
maxHeapSize: CARDINAL;
maxItemWords: CARDINAL;
occItemWords: CARDINAL;
putProc: PutProcType;
recordSize: CARDINAL;

RecordTooLong: PUBLIC ERROR = CODE;

BuildHeap: PROCEDURE =
  BEGIN
  L: CARDINAL;
  heapSize ← 0;
  MaintainHeap[];
  heapSize ← firstFreeEnt-1;
  L ← (heapSize/2)+1;
  WHILE L > 1 DO
    L ← L-1;
    SiftDown[L,heap[L]];
    ENDLOOP;
  RETURN;
  END;

BuildRuns: PROCEDURE =
  BEGIN
  --Continue reading and sorting, alternating in Fibonacci sequence, until the input is exhausted.
  A: CARDINAL;
  item: ItemHeaderHandle;
  i: CARDINAL;
  j: CARDINAL ← 1;
  LFile: FdHandle;
  NT: CARDINAL;
  level ← 1;
  DO OPEN files[j];
    IF level > 1 THEN dh.put[dh,EOR];  -- end-of-run marker
    FOR item ← GetHeap[], GetHeap[] UNTIL item = NIL DO
      dh.put[dh,item.len];
      [] ← StreamDefs.WriteBlock[dh,@item.rec,item.len];
      occItemWords ← occItemWords-item.len-SIZE[ItemLength]-FSPDefs.NodeOverhead;
      Free[item];
      ENDLOOP;
    dummyRuns ← dummyRuns-1;
    IF inputFinished AND (firstFreeEnt = 1) THEN EXIT;
    IF dummyRuns < files[j+1].dummyRuns THEN
      j ← j+1
```

```
    ELSE
      BEGIN
      j ← 1;
      IF dummyRuns = 0 THEN
        BEGIN
        level ← level+1;
        A ← files[1].totalRuns;
        FOR i IN[1..NFiles-1]
        DO
          LFile ← files[i];
          NT ← A+files[i+1].totalRuns;
          LFile.dummyRuns ←
          NT - LFile.totalRuns;
          LFile.totalRuns ← NT;
          ENDLOOP;
        END;
      END;
    BuildHeap[];
    ENDLOOP;
  FOR i IN [1..NFiles-1] DO OPEN files[i];
    dh.put[dh, EOR];
    dh.reset[dh];
    ENDLOOP;

  RETURN;
  END;

FreeAllocatedStuff:  PROCEDURE =
  BEGIN
  i: CARDINAL;
  FOR i IN[1..NFiles] DO
    IF files[i].buffer # NIL THEN Free[files[i].buffer];
    IF files[i].record # NIL THEN Free[files[i].record];
    Free[files[i]];
    ENDLOOP;
  IF BASE[heap] # NIL THEN Free[BASE[heap]];
  IF leftoverItem # NIL THEN Free[leftoverItem];
  EraseHeap[];
  RETURN;
  END;

GetHeap: PROCEDURE RETURNS[itemHP: ItemHeaderHandle] =
  BEGIN
  IF heapSize = 0 THEN RETURN[NIL];
  MaintainHeap[];
  itemHP ← heap[1];
  SiftDown[1,heap[heapSize]];
  firstFreeEnt ← firstFreeEnt-1;
  heap[heapSize] ← heap[firstFreeEnt];
  heapSize ← heapSize-1;
  RETURN;
  END;

Initialize: PROCEDURE [res, expected, max: CARDINAL] =
  BEGIN
  blockSize: INTEGER;
  heapPages,i: CARDINAL;
  res ← res + 92;  -- 82 for mesa(include bitmap), 10 for me
  heapPages ← AltoDefs.MaxVMPage - MAX[res,128];
  blockSize ← heapPages * AltoDefs.PageSize;
  InitHeap[heapPages];
  FOR i IN[1..NFiles] DO
    files[i] ← Alloc[SIZE[Fd]];
    files[i].buffer ← NIL;
    files[i].record ← NIL;
    ENDLOOP;
  bufferSize ← (blockSize)/NFiles - 100;
  recordSize ← IF bufferSize > LOOPHOLE[max,INTEGER] THEN max ELSE bufferSize;
  maxHeapSize ← (blockSize-recordSize)/(expected+3);  -- this 3 is magic
  maxItemWords ← blockSize-maxHeapSize-recordSize;
  occItemWords ← 0;
  RETURN;
  END;

MaintainHeap: PROCEDURE =
  BEGIN
```

```
  -- Fill the heap as full as possible
  itemHP: ItemHeaderHandle;
  IF inputFinished THEN RETURN;
  WHILE firstFreeEnt <= maxHeapSize DO
    -- Try adding another heap element
    IF NOT itemIsLeftOver THEN
      BEGIN
      leftoverItemLen ← getProc[leftoverItem];
      IF LOOPHOLE[leftoverItemLen,CARDINAL] > recordSize THEN
        ERROR RecordTooLong;
      IF leftoverItemLen = 0 THEN
        BEGIN
        inputFinished ← TRUE;
        EXIT;
        END;
      END;
    IF occItemWords >= maxItemWords THEN
      BEGIN
      itemIsLeftOver ← TRUE;
      EXIT;
      END;
    itemHP ← Alloc[leftoverItemLen+SIZE[ItemLength] !FSPDefs.NoRoomInZone =>
      BEGIN
      maxItemWords ← occItemWords - 100;
      itemIsLeftOver ← TRUE;
      --GOTO done;
      END];
    occItemWords ←
      occItemWords+leftoverItemLen+SIZE[ItemLength]+FSPDefs.NodeOverhead;
    itemHP.len ← leftoverItemLen;
    InlineDefs.COPY[leftoverItem,leftoverItemLen, @itemHP.rec];
    heap[firstFreeEnt] ← heap[heapSize+1];
    firstFreeEnt ← firstFreeEnt+1;
    heap[heapSize+1] ← itemHP;
    itemIsLeftOver ← FALSE;
    IF heapSize > 0 AND compareProc[@itemHP.rec,@heap[1].rec] = GT THEN
        BEGIN
        heapSize ← heapSize+1;
        SiftUp[];
        END;
    ENDLOOP;
  RETURN;
  END;

MergePass:  PROCEDURE =
  BEGIN
  dummiesThisPass: CARDINAL;
  lastFile: FdHandle;
  OFile: FdHandle;
  runNo: CARDINAL;
  runsThisPass: CARDINAL;

  OFile ← files[NFiles];
  lastFile ← files[NFiles-1];

  runsThisPass ← lastFile.totalRuns;
  dummiesThisPass ← lastFile.dummyRuns;

  -- FOR i IN[1..NFiles-2]
  dummiesThisPass ← MIN[dummiesThisPass,files[1].dummyRuns];

  OFile.totalRuns ← runsThisPass;
  OFile.dummyRuns ← dummiesThisPass;

  -- FOR i IN[1..NFiles-2]
  files[1].totalRuns ← files[1].totalRuns-runsThisPass;
  files[1].dummyRuns ← files[1].dummyRuns-dummiesThisPass;

  FOR runNo IN[dummiesThisPass+1..runsThisPass] DO
    MergeRun[OFile];
    ENDLOOP;
  IF level > 1 THEN
    BEGIN fd: FdHandle; i: CARDINAL;
    FlushBuffer[OFile];
    FOR i IN [NFiles-1..NFiles] DO OPEN files[i];
      dh.reset[dh];
```

```
      head ← 0;
      tail ← 0;
      ENDLOOP;
    fd ← files[NFiles];
    FOR i DECREASING IN (1..NFiles] DO
      files[i] ← files[i-1];
      ENDLOOP;
    files[1] ← fd;
    END;
  RETURN;
END;

MergeRun: PROCEDURE[OFile: FdHandle] =
  BEGIN
  -- Process a run.  Fill up the applicable records.
  i: CARDINAL;
  SR: CARDINAL;

  FOR i IN[1..NFiles-1] DO OPEN files[i];
    IF dummyRuns = 0 THEN
      [] ← ReadRecord[files[i]]
    ELSE
      BEGIN
      dummyRuns ← dummyRuns-1;
      endOfRun ← TRUE;
      END;
    ENDLOOP;

    DO
    SR ← 0;  -- selected record (which file is it from)
    FOR i IN[1..NFiles-1] DO OPEN files[i];
      IF (NOT endOfRun) AND (SR = 0 OR compareProc[
        record, files[SR].record] = LT) THEN SR ← i;
      ENDLOOP;
    IF SR = 0 THEN EXIT;  -- come back and fix this
    IF level = 1 THEN putProc[files[SR].record,files[SR].len]
    ELSE WriteRecord[OFile,files[SR].len,files[SR].record];
    files[SR].record ← NIL;  -- for cleanup guy
    [] ← ReadRecord[files[SR]];
    ENDLOOP;
  IF level > 1 THEN WriteRecord[OFile, -1, NIL];  -- end-of-run marker
  RETURN;
  END;

ReadRecord: PROCEDURE[file: FdHandle] RETURNS[BOOLEAN] =
  BEGIN
  itemLen: ItemLength;
  headIndex: INTEGER;
  IF file.head=LOOPHOLE[file.tail,CARDINAL] THEN FillBuffer[file,bufferSize];
  headIndex ← file.head;
  itemLen ← file.buffer↑[headIndex];
  file.head ← headIndex ← headIndex+1;
  IF itemLen < 0 THEN
    BEGIN
    file.endOfRun ← TRUE;
    RETURN[FALSE];
    END;
  IF headIndex+itemLen > file.tail THEN FillBuffer[file,bufferSize];
  headIndex ← file.head;
  file.record ← @file.buffer↑[headIndex];
  file.head ← headIndex+itemLen;
  file.len ← itemLen;
  file.endOfRun ← FALSE;
  RETURN[TRUE];
  END;

SiftDown: PROCEDURE[L: CARDINAL, K: ItemHeaderHandle] =
  BEGIN
  J: CARDINAL ← L;
  I: CARDINAL;
    DO
    I ← J;
    J ← J+J;
    IF J > heapSize THEN EXIT;
    IF J < heapSize THEN
      IF compareProc[@heap[J].rec,@heap[J+1].rec] > 0 THEN J ← J+1;
```

```
    IF compareProc[@K.rec,@heap[J].rec] <= 0 THEN EXIT;
    heap[I] ← heap[J];
    ENDLOOP;
  heap[I] ← K;
  RETURN;
  END;

SiftUp: PROCEDURE =
  BEGIN
  i: CARDINAL;
  j: CARDINAL ← heapSize;
  k: ItemHeaderHandle ← heap[heapSize];
  i ← j/2;
  WHILE i > 0 DO
    IF compareProc[@heap[i].rec,@k.rec] <= 0 THEN EXIT;
    heap[j] ← heap[i];
    j ← i;
    i ← j/2;
    ENDLOOP;
  heap[j] ← k;
  RETURN;
  END;

WriteRecord: PROCEDURE [file: FdHandle, itemLen: ItemLength,
  itemPtr: ItemHandle] =
  BEGIN
  buffer: ItemHandle ← file.buffer;
  tailIndex: INTEGER ← file.tail;
  IF tailIndex+(IF itemLen < 0 THEN 1 ELSE itemLen+1) > bufferSize THEN
    BEGIN
    FlushBuffer[file];
    tailIndex ← file.tail;
    END;
  buffer↑[tailIndex] ← itemLen;
  tailIndex ← tailIndex+1;
  IF itemLen >= 0 THEN
    BEGIN
    InlineDefs.COPY[itemPtr,itemLen,@buffer↑[tailIndex]];
    tailIndex ← tailIndex+itemLen;
    END;
  file.tail ← tailIndex;
  RETURN;
  END;

Sort: PUBLIC PROCEDURE [get: GetProcType, put: PutProcType,
  compare: CompareProcType, expectedItemSize: CARDINAL, maxItemSize: CARDINAL,
  reservedPages: CARDINAL] =
  BEGIN
  DefaultExpected: CARDINAL = 10;  -- words
  DefaultMax: CARDINAL = 1000;
  DefaultReserved: CARDINAL = 10;
  item: ItemHeaderHandle;
  fid: STRING = "SORT.SCRATCH0";
  i: CARDINAL;
  lastChar: CARDINAL ← fid.length-1;

  Initialize[IF reservedPages # 0 THEN reservedPages ELSE DefaultReserved,
    IF expectedItemSize # 0 THEN expectedItemSize ELSE DefaultExpected,
    IF maxItemSize # 0 THEN maxItemSize ELSE DefaultMax];
  getProc ← get;
  compareProc ← compare;
  putProc ← put;
  heap ← DESCRIPTOR[Alloc[maxHeapSize+1],maxHeapSize+1];
  firstFreeEnt ← 1;
  -- First, fill up the heap as much as possible and sort it.
  leftoverItem ← Alloc[recordSize];
  itemIsLeftOver ← FALSE;
  inputFinished ← FALSE;
  BuildHeap[];

  IF inputFinished THEN
    THROUGH [1..heapSize] DO
      item ← GetHeap[];
      put[@item.rec,item.len];
      ENDLOOP
  ELSE
```

```
    BEGIN
    FOR i IN[1..NFiles-1] DO OPEN StreamDefs, files[i];
      fid[lastChar] ← fid[lastChar]+1;
      dh ← NewWordStream[fid, Append+Write+Read];
      totalRuns ← 1;
      dummyRuns ← 1;
      ENDLOOP;
    files[NFiles].totalRuns ← 0;
    files[NFiles].dummyRuns ← 0;
    BuildRuns[];
    -- Put runs on input files 1..NFiles-1 so that they have Fibonacci relationship
    Free[leftoverItem];
    leftoverItem ← NIL;
    Free[BASE[heap]];
    heap ← DESCRIPTOR[NIL,0];
    IF level > 1 THEN
      BEGIN OPEN StreamDefs;
      fid[lastChar] ← fid[lastChar]+1;
      files[NFiles].dh ← NewWordStream[fid, Append+Write+Read];
      END;
    FOR i IN[1..NFiles] DO OPEN files[i];
      buffer ← Alloc[bufferSize];
      head ← 0;
      tail ← 0;
      ENDLOOP;
    -- Now carry out merge passes until the level has returned to zero.
    UNTIL level = 0 DO
      MergePass[];
      -- also cycles the files afterward if level>1
      level ← level-1;
      IF level = 1 THEN DeleteFile[files[NFiles].dh];  -- Output will go to the putItemParam routine
      ENDLOOP;
    FOR i IN [1..NFiles-1] DO
      DeleteFile[files[i].dh];
      ENDLOOP;
    END;
  FreeAllocatedStuff[];
  RETURN;
  END;

END...
```