

```

Nov 30, 02 20:57      distHeap.h      Page 1/2
/*****
*
* distHeap.h
*
* This file defines the interface for the distribution heap. A
* Distribution Heap is build of a number of levels (defined in
* level.h).
*
* by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
*****/
#ifndef DIST_HEAP
#define DIST_HEAP

#include<assert.h>
#include<iostream.h>
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<functional>
#include<vector>
#include<unistd.h>

#include "level.h"

#ifdef BOTTOM_HEAP
# include<queue>
#endif

// C defines the size of the smallest level in the heap. C is NOT
// allowed to be smaller than 9!!!!!!
// If C is smaller than 9, you will get unexpected errors, and wrong
// results!!
#ifdef LARGE_DATA
# define DEFAULT_C 9
#else
# define DEFAULT_C 9
#endif

namespace CPHSTL {

template<class T, class Compare = std::less<T> >
class priority_queue {
public:
    typedef int size_type;
    typedef T value_type;

private:
    Level<T,Compare>* firstLevel;
    T* start;
    T* space;
    T copy;

    Compare comp;
    int C;
    int size_;

#ifdef BOTTOM_HEAP
    // Make heaps to be used for insert and delete buffers.
    std::priority_queue<T, vector<T>, Compare>* insertHeap;
    std::priority_queue<T, vector<T>, Compare>* deleteHeap;
    int bottomHeapSize;
#else

```

```

Nov 30, 02 20:57      distHeap.h      Page 2/2
T* insertBuffer;
T* insertFirst;
T* insertLast;
int insertSize;

T* deleteBuffer;
T* deleteFirst;
T* deleteLast;
int deleteSize;
#endif

// Parameters used for global rebuilding.
int operationCount;
public:
    int nHalf;
private:
    void init();
    // Each time an operation is performed registerOperation() is
    // called. This increments the count of operations, and perform a
    // rebuild when needed.

    void registerOperation();
    void build(value_type* first, value_type* last);
    void fillDeleteBuffer();
    const value_type extractInternal();

public:
    ~priority_queue();
    priority_queue();
    priority_queue(int C_);

    // Build heap, with the values in [first, last).
    priority_queue(value_type* first, value_type* last, int C_);
    priority_queue(value_type* first, value_type* last);

    bool empty() const;

    size_type size() const;
    size_type realSize() const;

    void pop();

    // "e = a.extract;" is equal to: "e = a.top(); a.pop();"
    const value_type extract();
    const value_type top();
    void push(const value_type& elem_);

    void print();
};

}

#endif // DIST_HEAP

```

Nov 30, 02 21:07

distHeap.cpp

Page 1/11

```

/*****
 *
 * distHeap.cpp
 *
 * Implementation of the functions in the Distribution Heap. The
 * interface is defined in distHeap.h
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 *****/

#include "distHeap.h"
#include "../reverse_comp.h"

template<class T, class Compare>
void CPHSTL::priority_queue<T, Compare>::init() {
    if (C < 9){
        cerr << "ERROR: A Distribution Heap requires the parameter C to be larger than 8\n";
        exit(-1);
    }

    // Allocate space for one level (C). Since there is some rounding on
    // each computation, we need to adjust the computations to reflect
    // this.
    long long downSize = static_cast<long long>(2*ceil(pow(C, 0.666666)));
    long long noDown = static_cast<long long>(ceil(pow(C, 0.333333)));
    int toAlloc = 3*C + noDown + downSize + 1;

    start = new T[toAlloc];
    space = start;
    firstLevel = new Level<T,Compare>(C, start);

#ifdef BOTTOM_HEAP
    insertHeap = new std::priority_queue<T, std::vector<T>, Compare>();
    deleteHeap = new std::priority_queue<T, std::vector<T>, Compare>();
    bottomHeapSize = static_cast<int>(ceil(pow(C,0.66)/2));

    nHalf = C/2;
#else
    deleteSize = insertSize = static_cast<int>(ceil(pow(C,0.66)/2));
    insertBuffer = new T[insertSize + deleteSize];
    insertFirst = insertLast = insertBuffer;

    deleteBuffer = insertBuffer + insertSize;
    deleteFirst = deleteBuffer;
    deleteLast = deleteBuffer;

    nHalf = C/2;
#endif

    operationCount = 0;
    size_ = 0;
}

// Each time an operation is performed registerOperation() is called.
// This increments the count of operations, and perform a rebuild when
// needed.

```

Monday December 02, 2002

src/heap/distHeap/src/distHeap.cpp

Nov 30, 02 21:07

distHeap.cpp

Page 2/11

```

template<class T, class Compare>
void CPHSTL::priority_queue<T, Compare>::registerOperation() {
    operationCount++;

    if (operationCount < nHalf) {
        return;
    }
    // else: Perform rebuild!

    // create tmp buffer for elements
    int noElems = size();

    if (noElems == 0) {
        // No elems to rebuild!
        return;
    }

    T* tmp;
    tmp = new T[noElems];
    T* tmpLast = tmp;

#ifdef FAST_EXTRACT
    // Extract the elements from the old Distribution Heap, and sort
    // them.
    int insertSize;
    int deleteSize;

    // Move elements in insert and delete buffer.
#endif
#ifdef BOTTOM_HEAP
    insertSize = insertHeap->size();
    deleteSize = deleteHeap->size();

    while(!insertHeap->empty()) {
        *tmpLast++ = insertHeap->top();
        insertHeap->pop();
    }

    while(!deleteHeap->empty()) {
        *tmpLast++ = deleteHeap->top();
        deleteHeap->pop();
    }
#else
    insertSize = insertLast - insertFirst;
    deleteSize = deleteLast - deleteFirst;

    tmpLast = std::copy(insertFirst, insertLast, tmpLast);
    tmpLast = std::copy(deleteFirst, deleteLast, tmpLast);

    // Mark buffers empty
    insertLast = insertFirst;
    deleteLast = deleteFirst;
#endif

    int noElemsLeft = noElems - insertSize - deleteSize;

    // get elements from the levels.

```

2/36

Nov 30, 02 21:07

distHeap.cpp

Page 3/11

```

firstLevel->emptyInto(tmpLast);
assert(tmpLast == tmp + noElems);

// All elements are now in tmp (unsorted!)

//Sort tmp (in reverse order.)
sort(tmp, tmpLast, reverseComp<T, Compare>(comp));

#else // ifdef FAST_EXTRACT

// Use the old DH to extract the elems in sorted order.
// Extract elements
for (int i=0; i<noElems; i++) {
    *tmpLast++ = extractInternal();
}
// Since the elements are extracted, tmp is already sorted.

#endif // #ifdef FAST_EXTRACT

#ifdef BOTTOM_HEAP
// Rebuilds does not change element in bottom Heaps.
#else
delete[] insertBuffer;
delete[] space;
#endif

// remove the old heap.
delete firstLevel;

// Insert elements in a new heap. (New nHalf value is set by build.)
build(tmp, tmpLast);

// Reset counters
operationCount = 0;

delete[] tmp;
}

template<class T, class Compare>
CPHSTL::priority_queue<T, Compare>::~priority_queue() {
    delete firstLevel;

#ifdef BOTTOM_HEAP
    delete insertHeap;
    delete deleteHeap;
#else
    delete[] insertBuffer;
    delete[] space;
    // the deleteBuffer is allocated together with the insertBuffer, so
    // it is automatically deleted.
#endif
}

template<class T, class Compare>
CPHSTL::priority_queue<T, Compare>::priority_queue() {
    C = DEFAULT_C;
    init();
}

template<class T, class Compare>
CPHSTL::priority_queue<T, Compare>::priority_queue(int C_) {
    C = C_;

```

Nov 30, 02 21:07

distHeap.cpp

Page 4/11

```

    init();
}

template<class T, class Compare>
void CPHSTL::priority_queue<T, Compare>::build(value_type* first,
                                              value_type* last) {
    // Build assumes that [first, last) is sorted.
    init();
    size_ = last - first;

    // delete any old space and content.
    delete[] space;
    delete firstLevel;

    // Update nHalf.
    nHalf = max(C/2, size_/2);

    // Make precise computation of the needed space.
    int toAlloc = 0;

    long long currentSize = max(C, size_);
    do {

        // Due to rounding on the level, some more space are needed. This
        // is handled by repeating the same computations as when the level
        // is initialized.
        long long downSize = static_cast<long long>(
            2*ceil(pow(currentSize, 0.666666)));
        long long noDown = static_cast<long long>(
            ceil(pow(currentSize, 0.333333)));

        // Space needed:
        //      up buffer + no of down * size of down.
        toAlloc += currentSize + (noDown * downSize);

        // Compute size of next level.
        currentSize = static_cast<long long>(ceil(pow(currentSize, 0.666666)));

    } while (currentSize >= C);

    // Allocate the memory needed.
    start = new T[toAlloc];
    space = start;

    // Create the levels.

    Level<T,Compare>* previousLevel = 0;
    Level<T,Compare>* currentLevel;

    currentSize = max(C, size_);

    do {
        // create next level of size currentSize.
        currentLevel = new Level<T,Compare>(currentSize, start);

        // Make its nexLevel point to the level above.
        currentLevel->nextLevel = previousLevel;

        // Proceed with next level.

```

Nov 30, 02 21:07

distHeap.cpp

Page 5/11

```

// create next level of size currentSize.
previousLevel = currentLevel;

// Compute size of next level.
currentSize = static_cast<long long>(ceil(pow(currentSize, 0.666666)));
} while (currentSize >= C);

// currentLevel is the smallest level in the distribution heap.
firstLevel = currentLevel;

// Fill the elements into the levels, starting from the bottom.
firstLevel->fill(first, last);
}

template<class T, class Compare>
CPHSTL::priority_queue<T, Compare>::priority_queue(value_type* first,
                                                    value_type* last, int C_) {
    C = C_;
    // Sort in reverse order.
    sort(first, last, reverseComp<T, Compare>(comp));
    build(first, last);
}

template<class T, class Compare>
CPHSTL::priority_queue<T, Compare>::priority_queue(value_type* first,
                                                    value_type* last) {
    C = DEFAULT_C;
    // Sort in reverse order.
    sort(first, last, reverseComp<T, Compare>(comp));
    build(first, last);
}

template<class T, class Compare>
bool CPHSTL::priority_queue<T, Compare>::empty() const {
    return size_ == 0;
};

template<class T, class Compare>
CPHSTL::priority_queue<T, Compare>::size_type
CPHSTL::priority_queue<T, Compare>::size() const {
    return size_;
};

template<class T, class Compare>
void CPHSTL::priority_queue<T, Compare>::pop(){
    registerOperation();
    extract();
};

template<class T, class Compare>
const CPHSTL::priority_queue<T, Compare>::value_type
CPHSTL::priority_queue<T, Compare>::extract(){
    registerOperation();
    copy = extractInternal();
    size_--;

    return copy;
};

```

Nov 30, 02 21:07

distHeap.cpp

Page 6/11

```

template<class T, class Compare>
const CPHSTL::priority_queue<T, Compare>::value_type
CPHSTL::priority_queue<T, Compare>::extractInternal(){

#ifdef BOTTOM_HEAP
// Make sure that deleteBuffers are not empty.
if (deleteHeap->empty()) {
    fillDeleteBuffer();
}

if (!insertHeap->empty()) {
// If the deleteHeap is still empty, the heap is empty! Only the
// insertHeap contains elements.
if (deleteHeap->empty()) {
// Return element from insertHeap.
T tmp = insertHeap->top();
insertHeap->pop();
return tmp;
}

// There exists elements in both bottom heaps return the largest!
T deleteTop = deleteHeap->top();
T insertTop = insertHeap->top();

if (comp(insertHeap->top(), deleteHeap->top())) {
    deleteHeap->pop();
    return deleteTop;
}
else {
    insertHeap->pop();
    return insertTop;
}
}
else {
// insertHeap is empty. Return element from deleteBuffer.

if (deleteHeap->empty()) {
    cerr << " INTERNAL ERROR: Retrieving element from empty heap.\n";
    abort();
}
T tmp = deleteHeap->top();
deleteHeap->pop();
return tmp;
}

#else
// Make sure that deleteBuffers are not empty.
if (deleteFirst == deleteLast) {
    fillDeleteBuffer();
}

if (insertLast != insertFirst) {
// If the deleteBuffer is empty, the heap is empty! Only
// the insertBuffer contains elements.
if (deleteFirst == deleteLast) {
    insertLast--;
    return *insertLast;
}

// return the largest!
if (comp(*(insertLast - 1), *deleteFirst)) {
    return *deleteFirst++;
}
}

```

Nov 30, 02 21:07

distHeap.cpp

Page 7/11

```

    }
    else {
        insertLast--;
        return *insertLast;
    }
}
else {
    // insertBuffer is empty. Return element from deleteBuffer.

    if (deleteFirst == deleteLast) {
        cerr << " INTERNAL ERROR: Retrieving element from empty heap.\n";
        abort();
    }
    return *deleteFirst++;
}
#endif
}

template<class T, class Compare>
const CPHSTL::priority_queue<T, Compare>::value_type
CPHSTL::priority_queue<T, Compare>::top() {

    registerOperation();

#ifdef BOTTOM_HEAP
    // Make sure that deleteBuffers are not empty.
    if (deleteHeap->empty()) {
        fillDeleteBuffer();
    }

    if (!insertHeap->empty()) {
        // If the deleteHeap is still empty, the heap is empty! Only the
        // insertHeap contains elements.
        if (deleteHeap->empty()) {
            // Return element from insertHeap.
            return insertHeap->top();
        }

        // There exists elements in both bottom heaps. Return the largest!

        T deleteTop = deleteHeap->top();
        T insertTop = insertHeap->top();

        if (comp(insertHeap->top(), deleteHeap->top())) {
            return deleteTop;
        }
        else {
            return insertTop;
        }
    }
    else {
        // insertHeap is empty. Return element from deleteBuffer.

        if (deleteHeap->empty()) {
            cerr << " INTERNAL ERROR: Retrieving element from empty heap.\n";
            abort();
        }
        T tmp = deleteHeap->top();
        return tmp;
    }
}

```

Nov 30, 02 21:07

distHeap.cpp

Page 8/11

```

#else
    // Make sure that the delete-buffer is not empty.
    if (deleteFirst == deleteLast) {
        fillDeleteBuffer();
    }

    if (insertLast != insertFirst) {

        // If the deleteBuffer is still empty, the heap is empty! Only
        // the insertBuffer contains elements.
        if (deleteFirst == deleteLast) {
            return *(insertLast - 1);
        }

        // return the largest!
        if (comp(*(insertLast - 1), *deleteFirst)) {
            return *deleteFirst;
        }
        else {
            return *(insertLast - 1);
        }
    }
    else {
        // insertBuffer is empty. Return element from deleteBuffer.
        if (deleteFirst == deleteLast) {
            cerr << " INTERNAL ERROR: Retrieving element from empty heap.\n";
            abort();
        }
        return *deleteFirst;
    }
#endif
};

template<class T, class Compare>
void CPHSTL::priority_queue<T, Compare>::push(const value_type& elem_) {

    registerOperation();
    size_++;

#ifdef BOTTOM_HEAP
    // Insert element in the insertHeap.
    insertHeap->push(elem_);

    // If insert buffer is full!
    if (insertHeap->size() >= bottomHeapSize) {

        // We merge the elements from the two Heaps into sorted.

        T sorted[insertHeap->size() + deleteHeap->size()];
        T* sortedFirst = sorted;
        T* sortedLast = sorted;
        int insertSize = insertHeap->size();

        //Merge the constant of the two heaps.
        while (1) {
            if (insertHeap->size() != 0){
                if (deleteHeap->size() == 0 ||
                    comp(insertHeap->top(), deleteHeap->top())) {
                    // The insert element is the smallest.
                    *sortedLast++ = insertHeap->top();
                }
            }
            else {
                *sortedLast++ = deleteHeap->top();
            }
        }
    }
}

```

Nov 30, 02 21:07

distHeap.cpp

Page 9/11

```

        insertHeap->pop();
    }
    else {
        // The delete element is the smallest.
        *sortedLast++ = deleteHeap->top();
        deleteHeap->pop();
    }
}
else {
    // insert is empty, copy element from delete.
    if (deleteHeap->size() == 0) {
        // Done
        break;
    }
    *sortedLast++ = deleteHeap->top();
    deleteHeap->pop();
}
}

// Push insertSize elements.
firstLevel->push(sortedFirst, sortedFirst + insertSize);
sortedFirst += insertSize;
// Move the rest to the deletion-buffer.
while (sortedFirst != sortedLast) {
    deleteHeap->push(*sortedFirst++);
}
}

#else

// Insert element in the reverse sorted insert buffer.
T elem = elem_;

T* pSort = insertFirst;

// while elem is larger than the current element.
while(pSort != insertLast && comp(*pSort, elem)){
    pSort++;
}

// Now elem is larger than the next element. Insert elements, and
// move all other elements one slot.
while(pSort != insertLast) {
    // replace *pSort with elem
    T tmp = *pSort;
    *pSort++ = elem;
    elem = tmp;
}

// Insert the largest element in the next slot. (The first empty slot.)
*insertLast++ = elem;

// If insert buffer is full!
if (insertLast == insertFirst + insertSize) {
    // We merge the sorted insert and delete buffer, and push the
    // smallest elements into the heap.

    T sorted[insertSize + deleteSize];
    T* sortedFirst = sorted;
    T* sortedLast = sorted;

    //Merge the two buffers. Remember that they are sorted in

```

Nov 30, 02 21:07

distHeap.cpp

Page 10/11

```

//opposite directions.

while (1) {
    if (insertFirst != insertLast){
        if (deleteFirst == deleteLast ||
            comp(*insertFirst, *(deleteLast - 1))) {
            // The insert element is the smallest.
            *sortedLast++ = *insertFirst++;
        }
        else {
            // The delete element is the smallest.
            deleteLast--;
            *sortedLast++ = *deleteLast;
        }
    }
    else {
        // insert is empty, copy element from delete.
        if (deleteFirst == deleteLast) {
            // Done
            break;
        }
        deleteLast--;
        *sortedLast++ = *deleteLast;
    }
}

// Reset insert-buffer.
insertFirst = insertLast = insertBuffer;

// Push insertSize elements.
firstLevel->push(sortedFirst, sortedFirst + insertSize);
sortedFirst += insertSize;
// Move the rest to the deletion-buffer. In reverse order!
while (sortedFirst != sortedLast) {
    sortedLast--;
    *deleteLast++ = *sortedLast;
}
}
#endif

template<class T, class Compare>
void CPHSTL::priority_queue<T, Compare>::fillDeleteBuffer() {

#ifdef BOTTOM_HEAP
    // Compute size of heap!
    int heapSize = firstLevel->size();

    if (heapSize != 0){

        T* tmp = new T[bottomHeapSize];
        T* tmpFirst = tmp;
        T* tmpLast = tmp + bottomHeapSize;

        firstLevel->pull(tmpFirst, tmpLast);

        // Create a new deleteHeap with the new elements.
        delete deleteHeap;
        deleteHeap = new std::priority_queue<T, std::vector<T>, Compare>
            (tmpFirst, tmpLast);
    }
};

```

Nov 30, 02 21:07

distHeap.cpp

Page 11/11

```

#else
    // Compute size of heap!
    int heapSize = size_ - (insertLast - insertFirst) -
                    (deleteLast - deleteFirst);

    if (heapSize != 0){
        deleteFirst = deleteBuffer;
        deleteLast = deleteBuffer + deleteSize;

        firstLevel->pull(deleteFirst, deleteLast);
    };
#endif
}

template<class T, class Compare>
void CPHSTL::priority_queue<T, Compare>::print() {
    cout << "=====\n";

#ifdef BOTTOM_HEAP
    cout << "insertHeap:(" << insertHeap->size() << "):";
    if (!insertHeap->empty()) {
        cout << "Top" << insertHeap->top();
    }
    cout << endl;

    cout << "deleteHeap:(" << deleteHeap->size() << "):";
    if (!deleteHeap->empty()) {
        cout << "Top" << deleteHeap->top();
    }
    cout << endl;

#else
    cout << "insertBuffer: ";
    T* tmp = insertFirst;
    while (tmp != insertLast) {
        cout << *tmp++ << " ";
    }
    cout << endl;

    cout << "deleteBuffer: ";
    tmp = deleteFirst;
    while (tmp != deleteLast) {
        cout << *tmp++ << " ";
    }
    cout << endl;
#endif

    firstLevel->print();
    cout << "=====\n";
}

```

```

Nov 30, 02 21:10          level.h          Page 1/13
/*****
*
* level.h
*
* A distribution heap is constructed of a number of levels. This file
* defines a level, and the operations it provides.
*
* by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
*****/
#ifndef DIST_LEVEL
#define DIST_LEVEL

#include<utility>
#include<vector>
#include<functional>
#include<stdio.h>

#include<math.h>
#include"../reverse_comp.h"

#define OK 0
#define OUT_FULL 1

template<class T, class Compare = std::less<T> >
class Level {

    // Allocate space for the level.
    T* dataSpace;

    Compare comp;

    T* up;
    T* upEnd;

    long long upSize;
    long long downSize;
    long long noDown;

    // downs contains 2 pointers per down buffer the first is to the
    // first element, and the last is a one-past-the-end pointer.

    vector<T*> downs;
    long long downUsed;

public:
    long long levelSize;
    Level* nextLevel;

    Level(long long level, T*& start) {
        init(level, start);
    }

    Level(long long level, T*& start, Compare comp_) {
        comp = comp_;
        init(level, start);
    }

    ~Level() {
        if (nextLevel != 0) {
            delete nextLevel;
        }
    }
}

```

```

Nov 30, 02 21:10          level.h          Page 2/13
}

private:
void init(long long level, T*& start) {
    upSize = level;

    nextLevel = 0;
    levelSize = level;

    downSize = static_cast<long long>(2*ceil(pow(level, 0.666666)));
    noDown = static_cast<long long>(ceil(pow(level, 0.333333)));

    long long levelSize = upSize + noDown * downSize;

    dataSpace = start;

    up = upEnd = dataSpace;
    downUsed = 0;

    // Init the down buffers.
    T* free = dataSpace + upSize;

    for (long long i = 0; i < noDown; i++) {
        // Insert the first pointer for the down-buffer.
        downs.push_back(free);
        // Insert the last pointer.
        downs.push_back(free);
        free += downSize;
    }

    // Show how much of start has been used.
    start = free;
}

public:
void fill(T*& first, T*last) {
    // Insert elements in the level.

    // Fill down buffers with elements.
    int current = 0;
    while(current < noDown) {
        T* downFirst = downs[2*current];

        // Copy elements to down-buffer current.
        long long toCopy = min(static_cast<long long>(last-first), downSize/2);

        // Make room for pivot element.
        downFirst++;

        // Copy other elements.
        downFirst = copy(first, first + toCopy - 1, downFirst);
        first += toCopy - 1;

        // Copy pivot element.
        *(downs[2*current]) = *first++;

        downs[2*current + 1] = downFirst;
        downUsed++;

        if (first==last) {
            // All elements are copies, stop traversal.

```

Nov 30, 02 21:10

level.h

Page 3/13

```

    }
    // else: continue with the next down-buffer.
    current++;
}

nextLevel->fill(first, last);
}

void push(T* first, T* last) {
    // Sort input
    sort(first, last, reverseComp<T, Compare>(comp));
    // Distribute input on buffers.
    T* next = first;
    int currentDown = 0;

    if (downUsed == 0) {
        // Start using a down-buffer.
        downUsed = 1;

        T* downFirst = downs[0];
        T* downLast = downs[1];

        // downs[1] == pLast in first down buffer.
        // Make room for the pivot element.
        downLast++;

        // Copy the small elements to the buffer.
        while(next < last - 1 && downLast < (downFirst + downSize)){
            *downLast++ = *next++;
        }

        // Insert the pivot element (the last element in the sorted list).
        *(downs[0]) = *next++;

        // Write pointer back into downs!
        downs[1] = downLast;
    }

    T pivot = *downs[0];

    while(next < last) {
        if (comp(*next, pivot)) {
            // Proceed to next down buffer.
            if (currentDown == noDown - 1) {
                // Move the rest to the up buffer.
                while (next != last) {
                    *upEnd++ = *next++;
                    // Handle overflow in upbuffer
                    if (upEnd >= up+upSize) {
                        pushUpbuffer();
                    }
                }
            }
            else {
                // Use the next down buffer.
                currentDown++;

                if (currentDown >= downUsed) {
                    if (last - next >= downSize/4) {

```

Nov 30, 02 21:10

level.h

Page 4/13

```

        // The new buffer is empty, fill elements into it.

        // Mark the new buffer as used.
        downUsed++;

        T* downFirst = downs[2*currentDown];
        T* downLast = downs[2*currentDown + 1];

        // Make room for the pivot element.
        downLast++;

        // Copy the small elements to the buffer.
        int k = 0;
        while (next != last - 1 && downLast < (downFirst + downSize)){
            *downLast++ = *next++;
        }

        // Insert the pivot element.
        *downFirst = *next++;

        downs[2*currentDown + 1] = downLast;
    }
    else {
        // Not enough element for a new downbuffer, fill into up instead.
        while (next < last){
            *upEnd++ = *next++;
            // Handle overflow in upbuffer
            if (upEnd >= up+upSize) {
                pushUpbuffer();
            }
        }
    }
}

pivot = *downs[2*currentDown];
}

// We want to insert element in currentdown
else {
    if (downs[2*currentDown+1] < downs[2*currentDown] + downSize) {
        // Insert elem in currentDown
        *downs[2*currentDown+1]++ = *next++;
    }
    else {
        // the currentDown buffer is now full

        // to handle multiple instances of equal value, we move an
        // element to the first non-full buffer with that value. So
        // if the next buffer also has the elements value as pivot
        // element, we will place the element here.
        if (currentDown + 1 < downUsed &&
            !comp(*downs[2*(currentDown + 1)], *next)) {
            // increment currentDown, and continue.
            currentDown++;
            continue;
        }
        // else: We need to place the element in this buffer, so
        // split it!

        // If all downbuffers are in use
        if (downUsed == noDown) {

```


Nov 30, 02 21:10

level.h

Page 7/13

```

#endif

T partitionElement = *pNth;

T* readFirst = inFirst;
T* readLast = inLast;

// Handle the pivot and partition element.

// Copy pivot-elements
*inFirst++ = partitionElement;
*outFirst++ = pivot;
readFirst++;

// The new partition element is located in position pNth, we have
// now copied it to the down-buffer, so we remove it by copying
// the first element to its old position, thus removing it.
*pNth = *readFirst++;

// Distribute elements.

// If the two pivot elements have the same value.
if (!comp(pivot, partitionElement) &&
    !comp(partitionElement, pivot)) {

    // Split the buffer so that both buffers has the same number of
    // elements.

    // Since pivot == partition elements some of the elements with
    // that value goes into in, the rest, and the larger elements
    // goes to out.

    T* outFull = outFirst + (size/2) - 1;

    while(outFirst < outFull) {

        // if pivot == element
        if (!comp(pivot, *readFirst)) {
            if (outFirst == outStop) {

                // The out buffer is now full. STOP the copying, so that
                // caller can handle the situation.

                // Set inLast to point to first unused element, so caller
                // knows where to continue from.
                inLast = readFirst;
                return OUT_FULL;
            }
            *outFirst++ = *readFirst++;
        }
        else {
            *inFirst++ = *readFirst++;
        }
    }

    // The in-buffer is now filled with elements == pivot, the rest
    // (elements >= pivot) is copied to in.

    inFirst = copy(readFirst, readLast, inFirst);
}
else {

```

Nov 30, 02 21:10

level.h

Page 8/13

```

    pivot = partitionElement;

    while (readFirst != readLast) {
        if (comp(*readFirst, partitionElement)) {
            if (outFirst == outStop) {
                // The out buffer is now full. STOP the copying, so that
                // caller can handle the situation.

                // Set inLast to point to first unused element, so caller
                // knows where to continue from.
                inLast = readFirst;
                return OUT_FULL;
            }
            // else
            // Move elements to pNew
            *outFirst++ = *readFirst++;
        }
        else {
            *inFirst++ = *readFirst++;
        }
    }
}
return OK;
}

void pull(T* first, T*& last){
    int pullFoo = last - first;

    // Compute size of all down buffers.
    int downsSize = 0;

    for ( int i = 0; i < 3; i++) {
        downsSize += downs[2*i+1] - downs[2*i];
    }

    // Are there enough elements in the down buffers.
    if (downsSize < static_cast<long long>(ceil(0.75*downsSize))) {
        if (nextLevel != 0 && !nextLevel->empty()) {

            // Pull elements from level above.
            int toPull = levelSize;

            T in[toPull];
            T* inRead = in;
            T* inLast = in + toPull;

            nextLevel->pull(in, inLast);

            int inSize = inLast - in;

            // Sort up buffer, and make a copy.
            sort(up, upEnd, reverseComp<T, Compare>(comp));

            T upCopy[upEnd - up];
            T* upCopyFirst = upCopy;
            T* upCopyLast;

            // Move element from up-buffer to temp copy.
            upCopyLast = copy(up, upEnd, upCopyFirst);
            upEnd = up;

```

Nov 30, 02 21:10

level.h

Page 9/13

```

int mergedSize = inSize + downsSize;
T merged[mergedSize];
T* mergedFirst = merged;
T* mergedRead = merged;

// Move elements in down-buffers to merged.
for (int i=0; i<downUsed; i++) {
    mergedFirst = copy(downs[2*i], downs[2*i + 1], mergedFirst);
    // Markbuffer empty
    downs[2*i + 1] = downs[2*i];
}
downUsed = 0;

// Sort the elements from down-buffers.
sort(mergedRead, mergedFirst, reverseComp<T, Compare>(comp));

// Merge in and upCopy
for (int i = 0; i < inSize; i++) {
    if (upCopyFirst != upCopyLast) {
        if (inRead != inLast) {
            if (comp(*inRead, *upCopyFirst)) {
                *mergedFirst++ = *upCopyFirst++;
            }
            else {
                *mergedFirst++ = *inRead++;
            }
        }
        else {
            // in is empty, upCopy is not
            *mergedFirst++ = *upCopyFirst++;
        }
    }
    else {
        // upCopy is empty, in it not
        *mergedFirst++ = *inRead++;
    }
}

// The remaining elements are placed (unsorted) in the up-buffer.
upEnd = copy(upCopyFirst, upCopyLast, upEnd);
upEnd = copy(inRead, inLast, upEnd);

// Push the elements from merged.
push(merged, merged + mergedSize);
}
else {
    // There is no level above, empty the up buffer.
    if (up != upEnd) {
        T* upLast = upEnd;
        upEnd = up;
        push(up, upLast);
    }
}

// Extract the elements from the down-buffers.
for (int i = 0; i < downUsed; i++) {
    // We always operate on the first buffer, when done with it, we

```

Nov 30, 02 21:10

level.h

Page 10/13

```

// remove it from the front and start over.
T* downlFirst = downs[0];
T* downlLast = downs[1];

// sort the buffer
sort(downlFirst, downlLast, reverseComp<T, Compare>(comp));

// Extract elements from the down-buffer.
int toCopy = min(last - first, downlLast - downlFirst);
first = copy(downlFirst, downlFirst + toCopy, first);
downlFirst += toCopy;

// Are we done?
if (first == last) {

    // Make sure that the first buffer is not too empty.
    if (downlLast - downlFirst < downsSize/4) {
        // The first down-buffer is now to empty. Move the elements to
        // the next buffer.

        // Remove downl from the beginning of the list, and move it
        // to the end, as a now empty buffer.
        T* emptyDownFirst = downs[0];

        downs.erase(downs.begin(), downs.begin() + 2);
        downs.push_back(emptyDownFirst);
        downs.push_back(emptyDownFirst);

        downUsed--;

        if (downlFirst != downlLast) {
            // The first down-buffer holds few elements.
            // Move the elements to a tmp location.
            T tmp[downlLast - downlFirst];
            T* pTmp;

            pTmp = copy(downlFirst, downlLast, tmp);

            push(tmp, pTmp);
        }
    }
    else {
        // The first down-buffer is large enough. Move the remaining
        // elements to the front of the buffer.

        T* downlWrite = downs[0];

        // Move the pivot element to the first space.
        *downlWrite++ = *(downlLast - 1);

        // Copy elements to front of the buffer.
        downlWrite = copy(downlFirst, downlLast - 1, downlWrite);

        // Store Last pointer in downs.
        downs[1] = downlWrite;
    }

    // return, with the last pointer pointing to the last element found.
    last = first;
    return;
}

```

Nov 30, 02 21:10

level.h

Page 11/13

```

else {
    // We need to extract more elements.

    // down1 is empty, mark it in downs.
    T* emptyDownFirst = downs[0];

    downs.erase(downs.begin(), downs.begin() + 2);
    downs.push_back(emptyDownFirst);
    downs.push_back(emptyDownFirst);

    downUsed--;

    if (downUsed == 0) {
        // All down buffers are empty, and no more elements in levels above.
        // return the once we have extracted.
        last = first;
        return;
    }
}

// This buffer was just filled up until first, therefore
// last must be moved there too
last = first;
}

bool peep(T& res) {
    T* first = downs[0];
    T* last = downs[1];

    if (first == last) {
        // All down buffers are empty, look in the up buffer instead.
        first = up;
        last = upEnd;

        if (first == last) {
            return false;
        }
    }

    T max = *first++;
    while(first != last) {
        if (comp(max, *first++)) {
            max = *(first - 1);
        }
    }
    res = max;
    return true;
}

void pushUpbuffer() {
    // Make sure that next level exists
    if (nextLevel == 0) {
        cerr << "Next level needed, and does not exist.\n";
        abort();
    }

    // Push upbuffer to next level
    nextLevel->push(up, upEnd);
    upEnd = up;
}

```

Nov 30, 02 21:10

level.h

Page 12/13

```

}

bool empty() {
    // If one of the down-buffers contain elements return false.
    for (int i = 0; i < downUsed; i++) {
        if (downs[2*i] != downs[2*i+1]) {
            return false;
        }
    }

    // Are there elements in up?
    if (up != upEnd) {
        return false;
    }

    // Are there elements on the levels above?
    if (nextLevel != 0) {
        return nextLevel->empty();
    }

    return true;
}

int size() {
    int res = 0;
    for (int i = 0; i < downUsed; i++) {
        if (downs[2*i] != downs[2*i+1]) {
            res += downs[2*i+1] - downs[2*i];
        }
    }

    res += upEnd - up;

    if (nextLevel != 0) {
        res += nextLevel->size();
    }

    return res;
}

// emptyInto is used to empty the Distribution Heap during a
// rebuild. It moves the content into to, and recursively invoke
// itself on the levels above, to ensure that the entaiere
// Distribution Heap content is moved into to.
void emptyInto(T& to) {
    // Move content of down-buffers into to.
    int current = 0;
    while(current < noDown) {
        T* downFirst = downs[2*current];
        T* downLast = downs[2*current + 1];

        to = copy(downFirst, downLast, to);

        // Mark the buffer empty.
        downs[2*current + 1] = downFirst;
        current++;
    }

    // Move content of up-buffer into to.
    to = copy(up, upEnd, to);
}

```

Nov 30, 02 21:10

level.h

Page 13/13

```

// Mark the up-buffer empty.
up = upEnd;

// Recursively invoke emptyInto on the next level.
if (nextLevel != 0) {
    nextLevel->emptyInto(to);
}
}

// Print the level, used for debugging.
void print() {

    cout << "-----\n";
    cout << "UP (size: " << upEnd - up << "): ";

    T* here = up;
    while (here != upEnd) {
        cout << *here++ << " ";
    }

    cout << "\nDown: (DownSize: " << downSize << " upSize: " << upSize << ")\n";
    for (int i = 0; i < noDown; i++) {

        here = downs[2*i];
        T* stop = downs[2*i + 1];
        printf("%d:(size %d)", i, stop - here);

        while (here != stop) {
            cout << *here++ << " ";
        }
        cout << "\n";
    }

    if (nextLevel != 0) {
        nextLevel->print();
    }
};
};

#endif // DIST_LEVEL

```

Nov 30, 02 21:12

funnelHeap.h

Page 1/8

```

/*****
*
* funnelHeap.h
*
* Implementation of the Funnel Heap.
*
* by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
*****/
#ifndef FUNNEL_HEAP
#define FUNNEL_HEAP

#include<assert.h>
#include<iostream.h>
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<pair.h>
#include<functional>
#include<vector>
#include<unistd.h>

#include"buf.h"
#include"bin_merge.h"
#include"k_way_merger.h"
#include"out_of_store.h"
#include"link.h"

template<class T, class Compare = std::less<T> >
class priority_queue {

public:
typedef int size_type;
typedef T value_type;

typedef Buf<T> buf_type;
typedef BinMerger<T, Compare> bin_type;

private:
T* Iarray;
T* IarraySpace;

buf_type* I;
vector<Link<T, Compare>*> link;
size_type size_;
Compare comp;

void init() {
// Set handler function for new to write warning.
set_new_handler(out_of_store);

Iarray = new T[8];
// 8 should have been constantS[0], but this breaks g++ with
// internal compiler error. (Bug reported to g++).

IarraySpace = Iarray;
I = new buf_type(Iarray, constantS[0]);

// Create link 1.
link.push_back(new Link<T, Compare>(0, comp));

```

Nov 30, 02 21:12

funnelHeap.h

Page 2/8

```

size_ = 0;
}

public:
priority_queue() {
init();
}

// Build heap, with the values in [first, last).
priority_queue(value_type* first, value_type* last) {

// Creat basic structure and link 0.
init();

int currentLink = 0;

// While not done.
while (last > first) {

// If next free S-buffer is in next link create the link.
if (link[currentLink]->c >= constantK[currentLink]) {
appendNewLink(++currentLink);
}

int currentSize = min(last - first, constantS[currentLink]);

// sort elements for buffer currentS in link currentLink.
sort(first, first + currentSize, reverseComp<T, Compare>(comp));

// Create the new S-buffer
int c = link[currentLink]->c;
buf_type* old = link[currentLink]->S[c];

// Create the real buffer.
link[currentLink]->S[c] = new buf_type(first, first+currentSize);

// Sanity check!
assert(link[currentLink]->S[c]->capacity() == currentSize);

// Make the merger use the new buffer.
int destination = c;
int width = constantK[currentLink];
bin_type* parent = link[currentLink]->Merger;
buf_type* childBuf;

while (width > 2) {
// Find the child on the path to destination.
step(parent, destination, width, childBuf);
}

// Invariant: Parent is now the merger that uses s.
if (destination == 0) { // The buffer is the left-leaf.
parent->in1 = link[currentLink]->S[c];
}
else { // The buffer is the right-leaf.
parent->in2 = link[currentLink]->S[c];
}

// remove the old dummy buffer.
delete old;

// Increment c to indicate that c is now filled.

```

Nov 30, 02 21:12

funnelHeap.h

Page 3/8

```

    link[currentLink]->c++;

    // Move first to next unsorted position
    first += currentSize;
}

~priority_queue() {
    // delete IarraySpace;
    delete I;

    delete link[0]->V; // This recursively deletes the complete merger-tree!

    delete link[0]->A;

    // Delete the space allocated in the links.
    for (int i = 0; i < link.size(); i++) {
        delete link[i];
    }
}

bool empty() const {
    return (I->empty() && link[0]->A->empty() && link[0]->V->exhausted());
};

size_type size() const { return size_;};

const value_type& top() const {

    buf_type* A1 = link[0]->A;
    bin_type* V1 = link[0]->V;

    assert(!(I->empty() && A1->empty() && V1->exhausted()));

    if (A1->empty() && !V1->exhausted()) {
        V1->fill();
    }

    if (!I->empty()) {
        if (!A1->empty()) {
            // Elements in both
            if (comp(A1->peep(), I->peep())) {
                return I->peep();
            }
            else {
                return A1->peep();
            }
        }
        else { // A empty, I not
            return I->peep();
        }
    }
    else { // I empty, A not
        return A1->peep();
    }
};

void pop(){
    --size_;
    extractInternal();
};

```

Nov 30, 02 21:12

funnelHeap.h

Page 4/8

```

// "e = a.extract;" is equal to: "e = a.top(); a.pop();"
const value_type extract(){
    --size_;
    return extractInternal();
}

private:
const value_type extractInternal(){

    buf_type* A1 = link[0]->A;
    bin_type* V1 = link[0]->V;

    assert(!(I->empty() && A1->empty() && V1->exhausted()));

    if (A1->empty() && !V1->exhausted()) {
        V1->fill();
    }

    if (!I->empty()) {
        if (!A1->empty()) {
            // Elements in both
            if (comp(A1->peep(), I->peep())) {
                return I->extract();
            }
            else {
                return A1->extract();
            }
        }
        else { // A empty, I not
            return I->extract();
        }
    }
    else { // I empty, A not
        return A1->extract();
    }
};

void appendNewLink(int free) {
    // The empty link is not created yet. Create it.
    link.push_back(new Link<T, Compare>(free, comp));

    // Delete the old dummyBuffer
    delete link[free - 1]->V->in2;

    // and replace it with the new link.
    link[free - 1]->V->in2 = link[free]->A;
    link[free - 1]->V->merger2 = link[free]->V;
}

public:
void push(const value_type& elem) {
    ++size_;

    sortInsert(I, elem, comp);

    if (I->size() > 7) {
        // Perform a sweep
        sweep();
    }
};

```

Nov 30, 02 21:12

funnelHeap.h

Page 5/8

```

void sweep() {
    // Find the first link with a free S-buffer.
    int free = 0;
    while(free < link.size() && // The link is created
          link[free]->c >= constantK[free]) // All S-buffers are filled.
    {
        free++;
    }

    // Invariant: free is now index to the first link with empty S-buffers.

    if (free == link.size()) {
        appendNewLink(free);
    }

    // Invariant: the link <free> now exists.
    int c = link[free]->c;

    // Count the content of each buffer
    int Asize[free];

    for (int i = 0; i <= free; i++) {
        Asize[i] = link[i]->A->size();
    }

    int Bsize = link[free]->B->size();

    // We make room in sigma1 for the elements from A and B + the
    // maximal possible number of elements on the path true the
    // k-way merger.
    int sigmasize = Asize[free] + Bsize + kPathSize(constantK[free]);
    // Similarly we compute the size of sigma2 by finding the
    // maximal number of elements not in A and B.
    int sigma2size = size_ - (Asize[free] + Bsize);

    // Allocate space for sigma1 and sigma2
    T* sigmaArray = new T[sigmasize + sigma2size];
    T* sigmaSpace = sigmaArray;

    buf_type* sigma1 = new buf_type(sigmaArray, sigmasize);

    // Move elemets from A[free] to sigma1
    sigma1->append(link[free]->A);

    // Move elemets from B[free] to sigma1
    sigma1->append(link[free]->B);

    // Find the size of the buffers on the path in the k-way merger.
    int mergerBufSizes[constantK[free]];
    int mergerSize = 0; // Total size of buffers on the path.

    int destination = c;
    int width = constantK[free];
    bin_type* parent = link[free]->Merger;
    buf_type* childBuf;

    int level = 0;

    while (width > 1) {

```

Nov 30, 02 21:12

funnelHeap.h

Page 6/8

```

    // Find the child on the path to destination.
    step(parent, destination, width, childBuf);

    // record the size.
    mergerBufSizes[level] = childBuf->size();
    mergerSize += childBuf->size();

    // Add the elements to sigma1
    sigma1->append(childBuf);
    level++;
}

buf_type* sigma2 = new buf_type(sigmaArray, sigma2size);

// Mark A[free] as empty and V[free] as exhausted.
link[free]->A->cheatEmpty = true;
link[free]->V->cheatExhausted = true;

// Move elemets to sigma2
#ifdef FAST_EXTRACT
// Move elements from I into sigma2
sigma2->append(I);
assert(I->empty());

// Move elements from A1 into sigma2.

sigma2->append(link[0]->A);
assert((link[0]->A)->empty());

// traverse the merger tree and append all buffers to sigma2
if (free > 0 ) {
    if (link[0]->V != 0 && !(link[0]->V)->exhausted()){
        // (link[0]->V)->print();
        (link[0]->V)->emptyInto(sigma2);
    }
    assert((link[0]->V)->exhausted());
    assert(empty());
}

// Sort sigma2
sortBuf(sigma2, reverseComp<T, Compare>(comp));

# else
    while (!empty()) {
        sigma2->insert(extractInternal());
    }
# endif

assert(sigma2->size() <= sigma2size);

// Remove cheat marks
link[free]->A->cheatEmpty = false;
link[free]->V->cheatExhausted = false;

// We will implement sigma as a virtual buffer, that is only
// merged on demand, directly into the destinations.

```

Nov 30, 02 21:12

funnelHeap.h

Page 7/8

```

// Create sigma-merger.
bin_type sigmaMerger(sigma1, sigma2, 0, 0, 0, comp);

// Fill the buffers on path P.
for (int i = 0; i <= free; i++) {
    sigmaMerger.refill(link[i]->A, Asize[i]);
}

sigmaMerger.refill(link[free]->B, Bsize);

// Fill the buffers on p in the merger.
destination = c;
width = constantK[free];
parent = link[free]->Merger;
childBuf = 0;

level = 0;

while (width > 1) {
    // Going one level down in the tree.
    step(parent, destination, width, childBuf);

    // Refill buffer
    sigmaMerger.refill(childBuf, mergerBufSizes[level]);
    level++;
}

// Insert the rest in the first free S-Buffer.

// Make sure that the S-buffer exists.
allocateS(free, c);

// Sanity check: The buffer should now have room.
assert((link[free]->S[c])->capacity() >= sigmaMerger.size());

sigmaMerger.refill(link[free]->S[c], constantS[free]);

// Reset C for link < free.
for (int i = 0; i < free; i++) {
    link[i]->c = 0;
}

// increment C for link free.
link[free]->c++;

// Done using the sigmas, deallocate the space for them.
delete sigmaSpace;
// sigma1 and 2 is automatically removed by sigmaMergers destructor.
}

void allocateS(int free, int c) {
    // Make sure that this buffer is allocated.
    if ((link[free]->S[c])->capacity() == 0) {

        buf_type* old = link[free]->S[c];

        // Create the real buffer.
        T* Sarray = new T[constantS[free]];
        link[free]->S[c] = new buf_type(Sarray, constantS[free]);

        // Make the merger use the new buffer.
        int destination = c;

```

Nov 30, 02 21:12

funnelHeap.h

Page 8/8

```

int width = constantK[free];
bin_type* parent = link[free]->Merger;
buf_type* childBuf;

while (width > 2) {
    // Find the child on the path to destination.
    step(parent, destination, width, childBuf);
}

// Invariant: Parent is now the merger that uses s.
if (destination == 0) { // The buffer is the left-leaf.
    parent->in1 = link[free]->S[c];
}
else { // The buffer is the right-leaf.
    parent->in2 = link[free]->S[c];
}

// remove the old dummy buffer.
delete old;
};
};
#endif

```

Nov 30, 02 21:12

link.h

Page 1/2

```

/*****
 *
 * link.h
 *
 * A link contains different buffers and mergers. To construct a funnel
 * heap links are joint, and operations are propagating true the links.
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 *****/

// Tables of S and K
int constantS[7] = {8, 24, 120, 1080, 18630, 605880, 78158520};
int constantK[7] = {2, 4, 8, 16, 32, 128, 512};

template<class T, class Compare>
class Link {
public:
    typedef Buf<T> buf_type;
    typedef BinMerger<T, Compare> bin_type;

    int c;

    int id;

    buf_type* A;
    buf_type* B;

    buf_type** S;

    bin_type* V;
    bin_type* Merger;

    // Allocate and admin space for the buffers in the link.
    //
    // Invariant: start points to first unused slot, and is thus
    // updated when someone allocates space in it.
    //
    // Space is only used for deallocating in the destructor.
    T* start;
    T* space;

    Link(int id_, Compare& comp) {
        id = id_;
        // Allocate the space for the buffers in the link.
        // The size needed in the link is for buffers A, B and Merger.

        // Compute size of A and B.
        int bufSize = static_cast<int>(pow(static_cast<long double>(constantK[id]),
                                          3));
        int linkSize = static_cast<int>(2 * bufSize + kSize(constantK[id]));
        space = start = new T[linkSize];

        // Allocate A
        A = new buf_type(start, bufSize);

        // Allocate B
        B = new buf_type(start, bufSize);

        // Reserve space for the merger now. (The space is between
        // kStart and kEnd)
        T* kStart = start;

```

Nov 30, 02 21:12

link.h

Page 2/2

```

    start += kSize(constantK[id]);
    T* kEnd = start;

    // Allocate S buffers.
    S = new buf_type*[constantK[id]];

    // Create dummy S-buffers.
    for (int i = 0; i < constantK[id]; i++) {
        S[i] = new buf_type(start, 0);
    }

    // Create the K-way merger
    Merger = kWayMerger<T, Compare>(B, constantK[id], S, kStart, comp);

    //Sanity check: The merger should now have used the space reserved for it.
    assert(kStart == kEnd);

    // Create dummy buffer to pose as the buffer A from the link above.
    buf_type* emptyBuffer = new buf_type(start, 0);

    V = new bin_type(B, emptyBuffer, A, Merger, 0, comp);

    c = 0;

    // Sanity Check: The buffer allocated for the link should be used by now.
    assert(start <= space + linkSize);
}

~Link() {
    delete space;
    delete S;
};
};

```

Nov 30, 02 21:15

k_way_merger.h

Page 1/3

```

/*****
 *
 * k_way_merger
 *
 * Implementation of a K-way merger factory. This function creates and
 * allocates a K-way merger, and returns a pointer to the top BinMerger
 * in it.
 *
 * The space needed is allocated in the array specified by start. The
 * first element is placed in start. As a side effect, start is updated
 * so that start will always point to the first free location in start.
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 *****/
#ifdef FUNNEL_KMERGER
#define FUNNEL_KMERGER

;
#include<assert.h>
#include<math.h>

template<class T, class Compare>
BinMerger<T, Compare>* kWayMerger(Buf<T>* out, int k, Buf<T>** in,
                                  T*& start, Compare comp) {

    BinMerger<T, Compare>* sources[k];
    for (int i = 0; i<k; i++) {
        sources[i] = 0;
    }

    return kWayMerger(out, k, in, sources, start, comp);
}

template<class T, class Compare>
BinMerger<T, Compare>* kWayMerger(Buf<T>* out, int k, Buf<T>** in,
                                  BinMerger<T, Compare>** sources,
                                  T*& start, Compare comp) {

    // The base case
    if (k == 2) {

        return new BinMerger<T, Compare>(in[0], in[1], out,
                                          sources[0], sources[1], comp);
    }

    if (k < 2) {
        // Internal error!
        abort();
    }

    // Compute sizes
    double height = log(static_cast<long double>(k))/
                   log(static_cast<long double>(2));
    int heightInt = static_cast<int>(floor(height + 0.5));

    int bottomI = static_cast<int>(heightInt >> 1);
    int topI = heightInt - bottomI;

    int topK = 1 << topI;           // = 2^(topI)
    int bottomK = 1 << bottomI;     // = 2^(bottomI)

    // Compute sizes
    double height = log(static_cast<long double>(k))/
                   log(static_cast<long double>(2));
    int heightInt = static_cast<int>(floor(height + 0.5));

    int bottomI = static_cast<int>(heightInt >> 1);
    int topI = heightInt - bottomI;

    int topK = 1 << topI;           // = 2^(topI)
    int bottomK = 1 << bottomI;     // = 2^(bottomI)

```

Nov 30, 02 21:15

k_way_merger.h

Page 2/3

```

    int sizeofBuffers = static_cast<int>(ceil(pow(k,1.5)));

    // Reserve space for the top merger.
    T* kTopStart = start;
    start += kSize(topK);
    T* kTopEnd = start;

    // Create middle buffers.
    Buf<T>* midBuffers[topK];

    for (int i = 0; i < topK; i++) {
        midBuffers[i] = new Buf<T>(start, sizeofBuffers);
    }

    // Create bottom mergers.
    BinMerger<T, Compare>* bottomMergers[topK];

    for (int i = 0; i < topK; i++) {
        bottomMergers[i] = kWayMerger(midBuffers[i], bottomK,
                                       (in + (i*bottomK)), (sources + (i* bottomK)),
                                       start, comp);
    }

    // Create and return top merger
    BinMerger<T, Compare>* res = kWayMerger(out, topK, midBuffers,
                                             bottomMergers, kTopStart, comp);

    return res;
}

// Computes the size needed for buffers in a k-way merger.
int kSize(int k) {

    if (k<=2) {
        return 0; // A BinMerger does need extra buffers.
    }

    double height = log(static_cast<long double>(k))/
                   log(static_cast<long double>(2));

    int heightInt = static_cast<int>(floor(height + 0.5));

    int bottomI = static_cast<int>(heightInt >> 1);
    int topI = heightInt - bottomI;

    int topK = 1 << topI;           // = 2^(topI)
    int bottomK = 1 << bottomI;     // = 2^(bottomI)

    int sizeofBuffers = static_cast<int>(ceil(pow(k,1.5)));

    int totalMidSize = topK * sizeofBuffers;

    // The size is the middlebuffers + the size of the top-merger + the
    // size of the topK bottomMergers.
    return totalMidSize + kSize(topK) + (topK * kSize(bottomK));
}

// Computes the size of the buffers on a path from top to bottom.

```

Nov 30, 02 21:15

k_way_merger.h

Page 3/3

```

int kPathSize(int k) {
    if (k<=2) {
        return 0;
    }
    double height = log(static_cast<long double>(k))/
        log(static_cast<long double>(2));
    int topI = (int)(ceil(height/2));
    int topK = 1 << topI;          // = 2^(topI)
    int bottomI = static_cast<int>(floor(height/2));
    int bottomK = 1 << bottomI;    // = 2^(bottomI)
    int sizeofBuffers = static_cast<int>(ceil(pow(k,1.5)));

    // The size is the middlebuffers + the size of the top-merger + the
    // size of the bottomMerger on the path.
    return sizeofBuffers + kPathSize(topK) + kPathSize(bottomK);
}

// step() is used to traverse a K-Way Merger, from the root to the
// <destination>'th leaf. step() performs one traversal from
// parent, to the child on that path. When step returns all four
// parameters are updated to reflect this step down in the tree. So it
// is possible to call step() again immediately with the same
// parameters. You can also use <parent> to acces the bin-merger on the
// level, or childBuf to acces the output-buffer of that bin-merger.

template<class T, class Compare>
void step(BinMerger<T, Compare>*& parent, int& destination, int& width,
    Buf<T>*& childBuf ) {

    BinMerger<T, Compare>* child;

    // Find the child on the path.
    if (destination < (int)(width / 2)) {
        // destination is in the left half.
        child = parent->merger1;
        childBuf = parent->in1;
        width = (int)(width/2);
    }
    else { // destination is in the right half.
        child = parent->merger2;
        childBuf = parent->in2;
        width = (int)(width/2);
        destination -= width;
    }
    // Update parent to point at the next node.
    parent = child;
}

#endif

```

Nov 30, 02 21:15

bin_merge.h

Page 1/4

```

/*****
 *
 * bin_merge.h
 *
 * Implementation of a binary merger that merges two streams of sorted
 * data. It has two input buffers that it fetches the elements from, if
 * one of them runs empty, the corresponding input merger is invoked to
 * fill the buffer.
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 *****/
#ifdef FUNNEL_BIN
#define FUNNEL_BIN

template<class T, class Compare>
class BinMerger {
public:
    typedef Buf<T> buf_type;
    typedef BinMerger<T, Compare> bin_type;

    buf_type* in1;
    buf_type* in2;
    buf_type* out;
    bin_type* merger1;
    bin_type* merger2;
    Compare comp;

    bool cheatExhausted;

    ~BinMerger() {
        // When a bin merger is killed, it removes all the descendants, and
        // its input buffers. The output buffer is deleted by its parent.
        if (merger1 != 0)
            delete merger1;
        if (merger2 != 0)
            delete merger2;
        if (in1 != 0)
            delete in1;
        if (in2 != 0)
            delete in2;
    };

    BinMerger(buf_type* in1_, buf_type* in2_, buf_type* out_,
              bin_type* merger1_, bin_type* merger2_, Compare& comp_)
        : comp(comp_)
    {
        in1 = in1_;
        in2 = in2_;
        out = out_;
        merger1 = merger1_;
        merger2 = merger2_;
        cheatExhausted = false;
    }

    bool exhausted() {
        return (cheatExhausted ||
                ( in1->empty() && (merger1 == 0 || merger1->exhausted()) &&
                  in2->empty() && (merger2 == 0 || merger2->exhausted())));
    }
}

```

Monday December 02, 2002

src/heap/funnelHeap/src/bin_merge.h

Nov 30, 02 21:15

bin_merge.h

Page 2/4

```

int size() {
    int res = 0;

    res += in1->size();
    res += in2->size();

    if (merger1 != 0) {
        res += merger1->size();
    }

    if (merger2 != 0) {
        res += merger2->size();
    }

    return res;
}

void fill() {
    assert(!exhausted());

    bool exhausted1 = false;
    bool exhausted2 = false;

    while ( !(out->full()) ) {

        // Fill empty in buffers.
        if (!exhausted1 && in1->empty()) {
            if (merger1 == 0 || merger1->exhausted()) {
                exhausted1 = true;
            }
            else {
                merger1->fill();
            }
        }

        if (!exhausted2 && in2->empty()) {
            if (merger2 == 0 || merger2->exhausted()) {
                exhausted2 = true;
            }
            else {
                merger2->fill();
            }
        }

        // Perform one merge step;

        if (exhausted1 && exhausted2) {
            return;
        }
        if (exhausted1) {
            out->insert(in2->extract());
            continue;
        }
        if (exhausted2) {
            out->insert(in1->extract());
            continue;
        }

        if (comp(in2->peek(), in1->peek())) {
            out->insert(in1->extract());
        }
    }
}

```

22/36

Nov 30, 02 21:15

bin_merge.h

Page 3/4

```

    }
    else {
        out->insert(in2->extract());
    }
};

void refill(buf_type* dest, int n) {
    for (int i = 0; i < n; i++) {
        // You cant refill a buffer with more elemets than it can hold.
        assert(!(dest->full()));

        // Perform one merge step;
        if (in1->empty() && in2->empty()) {
            // No more input to merge.
            return;
        }
        if (in1->empty()) {
            dest->insert(in2->extract());
            continue;
        }
        if (in2->empty()) {
            dest->insert(in1->extract());
            continue;
        }

        if (comp(in2->peep(), in1->peep())) {
            dest->insert(in1->extract());
        }
        else {
            dest->insert(in2->extract());
        }
    }
}

// This function is used to empty a merger tree during a sweep
void emptyInto(buf_type* dest) {
    assert(dest);

    if (in1 != 0 && !in1->empty()){
        // Copy buffer to dest
        dest->append(in1);
    }

    if (merger1 != 0 && !merger1->exhausted()){
        // Traverse first child.
        merger1->emptyInto(dest);
    }

    if (in2 != 0 && !in2->empty()){
        // Copy buffer to dest
        dest->append(in2);
    }

    if (merger2 != 0 && !merger2->exhausted()){
        // Traverse second child.
        merger2->emptyInto(dest);
    }
}

```

Monday December 02, 2002

src/heap/funnelHeap/src/bin_merge.h

Nov 30, 02 21:15

bin_merge.h

Page 4/4

```

    }
}

void print() {
    cout << "-----\n" ;
    printInternal(0);
    cout << "-----\n" ;
}

void printInternal(int indent) {

    if (in1 != 0) {
        ident(indent);
        cout << "In1: ";
        in1->print();
        cout << endl;
    }

    if (in2 != 0) {
        ident(indent);
        cout << "In2: ";
        in2->print();
        cout << endl;
    }

    if (merger1 != 0) {
        ident(indent);
        cout << "merger1: ";
        cout << endl;
        merger1->printInternal(indent + 2);
    }

    if (merger2 != 0) {
        ident(indent);
        cout << "merger2: ";
        cout << endl;
        merger2->printInternal(indent + 2);
    }
}

void ident(int indent) {
    for (int i = 0; i < indent; i++) {
        cout << " ";
    }
}

};

#endif

```

23/36

Dec 01, 02 18:02

buf.h

Page 1/4

```

/*****
*
* buf.h
*
* Implementation of a circular buffer.
*
* by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
*****/

#ifndef FUNNEL_BUF
#define FUNNEL_BUF

#include<functional>
#include<assert.h>
#include<vector>

using namespace std;

template<class T>
class Buf {
    T* start;
    int max_size;

public:
    T* pFirst;
    T* pLast;
    int count;

    bool cheatEmpty;

    Buf(T& mem, int max_size_) {
        max_size = max_size_;
        pLast = pFirst = start = mem;
        count = 0;
        cheatEmpty = false;
        mem += max_size;
    }

    Buf(T* first, T* last) {
        count = max_size = last - first;
        start = pFirst = first;
        pLast = last - 1;
        cheatEmpty = false;
    };

    ~Buf() {};

    int capacity() {
        return max_size;
    }

    void increment(T* p) const {
        if (p == start + max_size - 1) {
            p = start;
        }
        else {
            ++p;
        }
    }

    bool full() {

```

Dec 01, 02 18:02

buf.h

Page 2/4

```

        return count == max_size;
    }

    int size() {
        return count;
    }

    bool empty() {
        return (count == 0 || cheatEmpty);
    }

    void insert(T elem) {
        assert(!full());
        *pLast = elem;
        increment(pLast);
        ++count;
    }

    void append(Buf<T>* in) {
        while(!in->empty()) {
            assert(!full());
            insert(in->extract());
        }
    }

    T extract() {
        assert(!empty());
        T tmp = *pFirst;
        increment(pFirst);
        --count;
        return tmp;
    }

    const T peep() {
        assert(!empty());
        return *pFirst;
    }

    void print() const{
        cout << "(" << count << "):";
        T* tmp = pFirst;
        for(int i = 0; i < count; i++) {
            cout << *tmp << " ";
            increment(tmp);
        }
    };

    template<class T, class Compare>
    void sortBuf(Buf<T>* buf, Compare comp) {
        assert(buf);
        // Ensure that the buffer is not wrapped.
        if (buf->pFirst > buf->pLast) {
            cerr << "Internal Error: Sorting Circular buffer.\n";
            abort();
        }
        sort(buf->pFirst, buf->pFirst + buf->count, comp);
    }

    template<class T, class Compare>
    void sortInsert(Buf<T>* buf, T elem, Compare comp) {

```

Dec 01, 02 18:02

buf.h

Page 3/4

```

assert(!buf->full());

T* pSort = buf->pFirst;

// while elem is smaller than the current element.
while(pSort != buf->pLast && comp(elem, *(pSort))) {
    buf->increment(pSort);
}

// Now elem is larger than the next element. Insert elem, and move
// all other elements one slot.
while(pSort != buf->pLast) {
    // replace *pSort with elem
    T tmp = *pSort;
    *pSort = elem;
    elem = tmp;
    buf->increment(pSort);
}

// Insert the largest element in the next slot. (The first empty.)
*(buf->pLast) = elem;
buf->increment(buf->pLast);
++(buf->count);
}

/////////////////////////////////////////////////////////////////
// VirtualBuf is a virtual buffer, that consists of a number of buffers.
/////////////////////////////////////////////////////////////////

template<class T>
class VirtualBuf {
    vector<Buf<T>* > subs;
    int first;
public:
    VirtualBuf() {
        first = 0;
    }

    ~VirtualBuf() {};

    void add(Buf<T>* buf) {
        subs.push_back(buf);
    }

    int capacity() {
        int res = 0;
        for(int i = 0; i < subs.size(); i++) {
            res += subs[i]->capacity();
        }
        return res;
    }

    int size() {
        int res = 0;
        for(int i = 0; i < subs.size(); i++) {
            res += subs[i]->size();
        }
        return res;
    }

    bool empty() {

```

Dec 01, 02 18:02

buf.h

Page 4/4

```

    bool res = true;
    for(int i = 0; i < subs.size(); i++) {
        res = (res && subs[i]->empty());
    }
    return res;
}

T extract() {
    while(first < subs.size() && subs[first]->empty()) {
        // increment first and report error if all subs are empty.
        first++;
        if (first >= subs.size()) {
            cerr << "Extract was called on an empty virtual buffer\n";
            abort();
        }
    }
    return subs[first]->extract();
}

const T& peep() {
    while(first < subs.size() && subs[first]->empty()) {
        // increment first and report error if all subs are empty.
        first++;
        if (first >= subs.size()) {
            cerr << "Extract was called on an empty virtual buffer\n";
            abort();
        }
    }
    return subs[first]->peep();
}
};

#endif

```

Nov 30, 02 21:16

funnelSort.h

Page 1/2

```

/*****
 *
 * funnelSort.h
 *
 * This .h file defines a cache-oblivious sorting method, based on
 * binary mergers.
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 *****/
#include "buf.h"
#include "bin_merge.h"
#include "k_way_merger.h"

#include "../reverse_comp.h"

#include <algorithm>
#include <math.h>
#include <cstdlib>

namespace FunnelSort {

template <class T, class Compare>
void sort(T* inFirst, T* inLast, Compare comp) {
    int size = inLast - inFirst;

    if (size <= 8) {
        // Base case sort with STL method.
        ::std::sort(inFirst, inLast, comp);
        return;
    }

    // Copy elems to tmp array
    T* tmp = new T[inLast - inFirst];
    T* first = tmp;
    T* last = copy(inFirst, inLast, first);

    typedef Buf<T> buf_type;

    // Find the size of the recursive buffers.
    long double Nroot = pow(size, 0.3333333);
    // Round Nroot to a the form 2^x
    long double ldTopHeight = log(Nroot)/log(static_cast<long double>(2));
    int topHeight = static_cast<int>(ldTopHeight);
    int topK = static_cast<int>(pow(static_cast<double>(2), topHeight));

    div_t foo = div(size, topK);
    int bottomSize = foo.rem == 0 ? foo.quot : foo.quot+1;
    int toSort = bottomSize;
    T* fromHere = first;

    buf_type* in[topK];

    // sort the recursive buffers.
    for (int i = 0; i < topK; i++) {
        if (fromHere + bottomSize > last) {
            toSort = last - fromHere;
        }
    }

```

Nov 30, 02 21:16

funnelSort.h

Page 2/2

```

    FunnelSort::sort<T, Compare>(fromHere, fromHere + toSort, comp);

    // create buffer to represent this sorted input.
    in[i] = new buf_type(fromHere, fromHere + toSort);

    int sorted = toSort;

    fromHere+= toSort;
}

// Allocate space for K-merger.
T* start = new T[kSize(topK)];
T* space = start;

// Allocate space for the output.
T* outStart = new T[size];
T* space2 = outStart;
buf_type out(outStart, size);

// Create the main merger.
BinMerger<T, reverseComp<T, Compare> >* top;
top = kWayMerger<T, reverseComp<T, Compare> >(&out, topK, in, start,
                                             reverseComp<T, Compare>(comp));

top->fill();
// Copy result back in the range.
for (int i = 0; i < size; i++) {
    *(inFirst++) = out.extract();
}

//Deallocate the space allocated.
delete top; // Also removes the input buffers.
delete[] space;
delete[] space2;
delete[] tmp;
}

} // end namespace

```

```

Nov 30, 02 21:19      heapTest.h      Page 1/6
/*****
*
* heapTest.h
*
* This file defines the performance measurements that are available
* for comparing heaps. This file provides the parsing of
* parameters. The primitives for the individual measurements are
* included from src/template/main.h
*
* by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
*****/
#include <stdio.h>
#include <main.h>
#include <assert.h>

#include "out_of_store.h"

#ifdef DIST
# define NAME CPHSTL::
#else
# define NAME
#endif

// By defining LARGE_DATA, the benchmrks are performed on large
// elements.
#ifdef LARGE_DATA
#include "largeType.h"
#define TYPE LargeData
#else
#define TYPE int
#endif

#ifdef DOTTRACE
#define CPROF "profile.dat"
#include <newprof.h>
#endif

NAME priority_queue<TYPE>* heap;

int workingOn;
int seed = 0;

using namespace std;

inline void ins() {
    heap->push(TYPE(rand()));
}

inline void push(int n) {
    // Insert n elements in the pd
    for (int i = 0; i < n; i++) {
        ins();
    }
}

void pushTest(int n) {
    srand(seed);
    // Insert n elements in the pd
    for (int i = 0; i < n; i++) {
        int foo = rand()%10;
        workingOn = i;
        heap->push(foo);
    }
}

```

```

Nov 30, 02 21:19      heapTest.h      Page 2/6
    }
    assert(heap->size() == n);
}

inline TYPE del() {
# ifdef EXTRACT_HEAP
    TYPE tmp = heap->extract();
# else
    TYPE tmp = heap->top(); heap->pop();
# endif
    return tmp;
}

inline void pop(int n) {
    // Insert n elements in the pd
    for (int i = 0; i < n; i++) {
        del();
    }
}

enum testType {full, insert, extract, build, buildCorrect,
               correct, complex, mono};

testType test;
int n;
int a;
int b;
int c = 0;

void doIt(){
    TYPE* bar = 0;

    if (test == build || test == buildCorrect || test == extract){
        bar = new TYPE[n];
        for (int i = 0; i < n; i++) {
            bar[i] = TYPE(rand());
        }

        if (test != extract) {
            startMeasure();
        }

        // The DIST_HEAP are parametrized with a constant c. Parse it on
        // if it is defined on the command line.
#ifdef DIST_HEAP
        if (c != 0) {
            heap = new NAME priority_queue<TYPE>(bar, bar + n, c);
        }
        else
#endif
        heap = new NAME priority_queue<TYPE>(bar, bar + n);

        if (test != extract) {
            stopMeasure();
        }
    }
    else {
#ifdef DIST_HEAP
        if (c != 0) {
            heap = new NAME priority_queue<TYPE>(c);
        }
        else

```

Nov 30, 02 21:19

heapTest.h

Page 3/6

```

#endif
    heap = new NAME priority_queue<TYPE>;
}
TYPE prev, tmp;
int prevSize;
int max = RAND_MAX;

switch (test) {
case (full) :
    startMeasure();
    push(n);
    pop(n);
    stopMeasure();
    break;
case (extract) :
    // The elements are inserted by build!
    startMeasure();
    pop(n);
    stopMeasure();
    break;
case (insert) :
    startMeasure();
    push(n);
    stopMeasure();
    break;
case (mono) :
#ifdef LARGE_DATA
    cerr << "Mono not supported on LARGE_DATA\n";
    abort();
#else
    startMeasure();

    // Insert initial data.
    push(n);
    assert( heap->size() == n);
    for (int i = 0; i<n; i++) {

        // Gennerate a random number in the range [min, RAND_MAX)
        int baz = rand();

        double r = ( (double)rand() / (double)(RAND_MAX) );

        int x = (static_cast<int>(r * max));
        // x is a random int in the range [0,max)

        assert(x <= max);

        heap->push(TYPE(x));

        // extract element
        int tmp = max;
        max = del();
    }

    // Pop the remaining elements.
    pop(n);

    stopMeasure();
#endif
break;
case (complex) :

```

Monday December 02, 2002

Nov 30, 02 21:19

heapTest.h

Page 4/6

```

#   ifndef LEDASM_HEAP
    cerr << "LEDA-SM does not support the complex test.\n";
    exit(-1);
#   endif
    startMeasure();
    for (int i = 0; i<b; i++) {
        ins();
        for (int j=0; j<a; j++) {
            del();
            ins();
        }

        for (int i = 0; i<b; i++) {
            del();
            for (int j=0; j<a; j++) {
                ins();
                del();
            }
        }
        stopMeasure();
        break;
    case (correct) :
        assert(heap->size() == 0);
        pushTest(n);

        // Extract, and compare the elements
        prev = del();

        for (int i = 0; i < n - 1; i++) {
            TYPE tmp = del();

#ifdef REVERSEHEAP
            if (prev > tmp)
#else
            if (prev < tmp)
#endif
            {
                cerr << "ERROR\n" << endl;

                // Leda-sm changes definition of strings, so avoid this in leda-sm.
                #ifndef LEDASM_HEAP
                    cerr << "Latest element: " << prev << " nowigot: " << tmp
                        << " iis: " << i << endl;
                #endif
            }
            exit(-1);
        }
        if ((i + 2 + heap->size()) != n) {
            cerr << "Heap size: " << heap->size() << " I: " << i << endl;
            cerr << "The size of the heap is not correct!\n";
            abort();
        }

        prev = tmp;
    }
    assert(heap->empty());
    cerr << "Result was OK!!!\n";
    break;
    case (buildCorrect) :
        // Extract, and compare the elements
        prev = del();
        for (int i = 0; i < n - 1; i++) {

```

src/heap/heapTest.h

28/36

Nov 30, 02 21:19

heapTest.h

Page 5/6

```

    tmp = del();
    if (prev < tmp) {
        cerr << "ERROR\n" << endl;
// Leda-sm changes definition of strings, so avoide this in leda-sm.
#ifdef LEDASM_HEAP
        cerr << "Latest element: " << prev << " nowigot: " << tmp
            << " iis: " << i << endl;
#endif
        exit(-1);
    }
    prev = tmp;
}
cerr << "Result was OK!!!\n";
break;
}
// DO NOT REMOVE
#ifdef DOTRACE
// Write the prof trace to the file.
writetofile();
#endif
if (bar != 0) {
    delete[] bar;
}
delete heap;
}

void use(char* command) {
    cerr << " Usage: " << command <<
        " [-s <number of operations to perform>] \n"
        << " ((-f|-e|-i|-c|-u))(-k <a> <b>)) [-d <seed for rand>]" <<
        " [-C <c value for distHeap>]\n" <<
        " -e = Measure extractions, -i = Measure insertions, " <<
        " -f = Measure both, -u measure Building\n" <<
        " -c = check correctness (Not a real measurement)\n" <<
        " -bc = check build correctness (Not a real measurement)\n" <<
        " -k = will perform complex (I(CI)^a)^b (D(ID)^a)^b";
}

void warn() {
    cerr << "WARNING: Multiple tests specified, using the last one.\n";
}

int main (int argc, char** argv) {
    // Set handler function for new to give error on exhaustion of memory.
    set_new_handler(out_of_store);

    // Default values!
    n = 10;
    test = full;
    bool testSet = false;

    // Parse parameters.
    int i = 1;
    while (i < argc) {
        if (!strcmp(argv[i], "-s")) {n = atoi(argv[++i]); ++i;}
        else if (!strcmp(argv[i], "-d")) {seed = atoi(argv[++i]); ++i;}
        else if (!strcmp(argv[i], "-C")) {c = atoi(argv[++i]); ++i;}
        else if (!strcmp(argv[i], "-f")) {
            if (testSet){warn();}; testSet = true; test = full; ++i;}
        else if (!strcmp(argv[i], "-e")) {
            if (testSet){warn();}; testSet = true; test = extract; ++i;}
        else if (!strcmp(argv[i], "-i")) {

```

Nov 30, 02 21:19

heapTest.h

Page 6/6

```

        if (testSet){warn();}; testSet = true; test = insert; ++i;}
    else if (!strcmp(argv[i], "-m")) {
        if (testSet){warn();}; testSet = true; test = mono; ++i;}
    else if (!strcmp(argv[i], "-u")) {
        if (testSet){warn();}; testSet = true; test = build; ++i;}
    else if (!strcmp(argv[i], "-k")) {
        if (testSet){warn();};
        testSet = true;
        test = complex;
        ++i;
        a = atoi(argv[i]);
        ++i;
        b = atoi(argv[i]);
        ++i;
    }
    else if (!strcmp(argv[i], "-bc")) {
        if (testSet){warn();}; testSet = true; test = buildCorrect; ++i;}
    else if (!strcmp(argv[i], "-c")) {
        if (testSet){warn();}; testSet = true; test = correct; ++i;}
    else {
        cerr << "Unknown command: '" << argv[i] << "' " << endl; use(argv[0]);
        exit(-1);
    }
}
measure();
}

```

Nov 30, 02 21:18

reverse_comp.h

Page 1/1

```
/*
 * reverse_comp.h
 *
 * ReverseCompare is used to reverse a comparison-function. This is
 * e.g. used when wanting to sort elements in decending order, using
 * only the less() function.
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 */
#ifndef REVERSE_COMPARE
#define REVERSE_COMPARE

template<class T, class Compare>
class reverseComp{
    Compare comp;
public:
    bool operator()(const T& x, const T& y) {
        return comp(y,x);
    };

    reverseComp(Compare comp_)
        : comp(comp_) {};
};

#endif
```

Nov 30, 02 21:18

largeType.h

Page 1/2

```

/*****
 *
 * large_type.h
 *
 * Definition of 100 byte large data type, used to benchmark the
 * implementations.
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 *****/
#ifdef LARGEDATADEFINITION
#define LARGEDATADEFINITION

#include <stdio.h>
#include <iostream.h>
#include <string.h>

#include <fstream>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <cstdlib>
#include <strstream>
#include <assert.h>

using namespace std;

struct LargeData{
public:
    int key1, key2, key3;
    char empty[88];

    LargeData(int a): key1(a), key2(rand()), key3(rand()) {};
    LargeData(){};

    bool operator< (LargeData other) const {
        return key1 < other.key1 ||
            (key1 == other.key1 && (key2 < other.key2 ||
                (key2 == other.key2 && key3 < other.key3)));
    }

    // Sanders heap require all the compare-functions, so we have to
    // define them, for it to compile. These are not used by other
    // heaps.
    bool operator< (const LargeData* const other) const {
        return key1 < other->key1 ||
            (key1 == other->key1 && (key2 < other->key2 ||
                (key2 == other->key2 && key3 < other->key3)));
    }

    bool operator> (LargeData other) const {
        return other < this;
    }

    bool operator>= (LargeData other) const {
        return !( *this < other);
    }

    bool operator<= (LargeData other) const {
        return !(other < this);
    }
}

```

Nov 30, 02 21:18

largeType.h

Page 2/2

```

    bool operator!= (LargeData other) const {
        return !(*this == other);
    }

    bool operator== (LargeData other) const {
        return !(*this < other || other < this);
    };
};

// Leda-sm changes definition of strings, so avoide this in leda-sm.
#ifdef LEDASM_HEAP

ostream&
operator<< (ostream& out, LargeData& data) {
    out << data.key1 << " " << data.key2 << " " << data.key3; return out;
}

#endif

#endif // ifndef LARGEDATADEFINITION

```

Nov 30, 02 21:18

empty.h

Page 1/1

```

/*****
 *
 * empty.cpp
 *
 * The "empty priority queue" does not do anything in any of the
 * functions. It is used as a reference to determine the cost of the
 * various benchmarks themselves.
 *
 * by: Jesper Holm Olsen and Søren Skov 2002 University of Copenhagen.
 *****/

#include<functional>

using namespace std;

template<class T, class Compare = std::less<T> >
class priority_queue {
public:
    typedef int size_type;
    typedef T value_type;

    int size_;

public:
    priority_queue() {
        size_ = 0;
    }

    // Build heap, with the values in [first, last).
    priority_queue(value_type* first, value_type* last) {
        size_ = last - first;
    }
    ~priority_queue() {
    }

    bool empty() const {
        return size_ == 0;
    };

    size_type size() const { return size_;}

    const value_type& top() const {
        return T();
    };

    void pop(){
        size_--;
    };

    // "e = a.extract;" is equal to: "e = a.top(); a.pop();"
    const value_type extract(){
        size_--;
        return T();
    }
public:
    void push(const value_type& elem) {
        size_++;
    };
};

```

Nov 30, 02 21:18

alloc.cpp

Page 1/2

```

/*****
*
* alloc.cpp
*
* Simulates the allocation of levels, and write the number of
* sucessful elements allocated to a file.
*
* by: Jesper Holm Olsen and Soren Skov 2002 University of Copenhagen.
*****/
#include<assert.h>
#include<iostream.h>
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<functional>
#include <iostream>
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <cstdlib>
#include <strstream>

#include<unistd.h>

#include "out_of_store.h"

#include "../distHeap/src/level.h"
#include "../distHeap/src/distHeap.h"

#include "../largeType.h"

int C = 0;

using namespace std;

void doIt(){

#ifdef LARGE_DATA
    typedef LargeData T;
#else
    typedef int T;
#endif

    typedef std::less<int> Compare;

    priority_queue<T>* pq = new priority_queue<T>();
    // Open file for writing the results.
    ofstream outFile;

    Level<T, Compare>* foo[20];
    int i = 1;

    cout << "C: " << C << endl;

    long long levelSize = C;
    long long total = 0;
    long long absoluteMax = 4000000000 / sizeof(T);

    while (1) {

```

Nov 30, 02 21:18

alloc.cpp

Page 2/2

```

    // allocate a level
    foo[i] = new Level<T, Compare>(levelSize);

    total += 3*levelSize;

    if (total > absoluteMax) {
        // Memoryspace exhausted.
        exit(-1);
    }

    cout << i << " " << total << endl;

    outFile.open("timeit.dat");
    outFile << total << endl;
    outFile.close();

    levelSize = static_cast<long long>(ceil(pow(levelSize,1.5)));
    i++;
}

int main (int argc, char** argv) {
    // Parse parameters.
    int i = 1;
    while (i < argc) {
        if (!strcmp(argv[i],"-C")) {C = atoi(argv[++i]);++i;}
        else {++i;}
    }
    doIt();
}

#include "../distHeap/src/distHeap.cpp"

```

```

Nov 30, 02 21:19      main.h      Page 1/4
#ifdef SYSTEM_MAIN
#define SYSTEM_MAIN

// #include <string>
#include <iostream>
#include <iostream.h>
#include <fcntl.h>
#include <fstream.h>
#include <stdio.h>
#include <algorithm>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <cstdlib>
#include <strstream>
#include <assert.h>

#ifdef DOPROC
// Include /proc functions.
# ifdef SPARC
#   include <sys/resource.h>
#   include <sys/procfs.h>
# else
#   include <sys/resource.h>
# endif
#endif

void doIt();

using namespace std;

#ifdef DOPAPI
#include <papi.h>

// Determine which tests to perform on which architectures.
#ifdef SPARC
# define LENGTH 5
int events[LENGTH] = {
    PAPI_L1_LDM,
    PAPI_L1_STM,
    PAPI_L1_ICH,
    PAPI_L1_ICA,
    PAPI_L2_TCM};
int todoses[LENGTH] = {1,1,1,1,1};
#else
#ifdef AMD
# define LENGTH 5
int events[LENGTH] = {
    PAPI_L1_DCM,
    PAPI_L1_DCA,
    PAPI_L2_DCM,
    PAPI_L2_DCA,
    PAPI_TLB_DM};
int todoses[LENGTH] = {4,0,0,0,1};
#else
# define LENGTH 6
int events[LENGTH] = {
    PAPI_L1_DCM,
    PAPI_L1_TCM,
    PAPI_L2_TCM,
    PAPI_L1_DCA,
    PAPI_L2_DCA,

```

```

Nov 30, 02 21:19      main.h      Page 2/4
    PAPI_L1_DCH};
int todoses[LENGTH] = {1,1,1,1,1,1};
#endif
#endif

long_long results[LENGTH] = {};
const PAPI_preset_info_t *info = NULL;
int done;
int todo;
#endif

#ifdef DOTIME
#include <sys/time.h>
// vars for timemesurement.
double beginMS;
double beginS;
double endMS;
double endS;
struct timeval tv;
#endif

#ifdef DOPROC
long majorPageFaults;
long majorPageFaultsAfter;

void startReadPageFaults(){

#ifdef SPARC
int fd;
char proc[100];
prusage_t prusage;

sprintf(proc, "/proc/%d", getpid());
if ((fd = open(proc, O_RDONLY)) == -1) {
    cerr << "Could not open /proc\n";
    exit(-21);
}
if (ioctl(fd, PIOCUSAGE, &prusage) == -1) {
    cerr << "Could not read proces' usage\n";
    exit(-22);
}

majorPageFaults = prusage.pr_majf;
#else
rusage      usages;
getrusage(RUSAGE_SELF, &usages);

majorPageFaults = usages.ru_majflt;
#endif
}

void stopReadPageFaults(){
ofstream procOut;
procOut.open("proc.dat");

#ifdef SPARC
int fd;
char proc[100];
prusage_t prusage;

if ((fd = open("/proc/self", O_RDONLY)) == -1) {
    cerr << "Could not open /proc\n";

```

Nov 30, 02 21:19

main.h

Page 3/4

```

    exit(-21);
}
if (ioctl(fd, PIOCUSAGE, &prusage) == -1) {
    cerr << "Could not read proces' usage\n";
    exit(-22);
}

majorPageFaultsAfter = prusage.pr_majf;

#else
usage usages;
getrusage(RUSAGE_SELF, &usages);
majorPageFaultsAfter = usages.ru_majflt;
#endif
int diff = majorPageFaultsAfter - majorPageFaults;
// printf("%d - %d = %d\n", majorPageFaultsAfter, majorPageFaults, diff);

cout << "Major page faults: " << diff << endl;
procOut << diff;
}

#endif

void measure() {

#ifdef DOPAPI
done = 0;
while(done < LENGTH) {
    // Find out how many can be done at the same time !!!!!!!
    todo = todos[done];

    doIt();
    done += todo;
}

// Write PAPI statistic to file!
if ((info = PAPI_query_all_events_verbose()) == NULL) {
    cerr << "query failed\n";
    exit(-3);
}

ofstream papiOut;
papiOut.open("papi.dat");
if (!papiOut) {
    cerr << "Could not open papi.dat\n";
    exit(-10);
}
for (int i = 0; i < LENGTH; i++) {
    char foo[200];

    sprintf(foo, "%s\t", info[events[i]].event_name);
    cout << foo << results[i] << endl;
    papiOut << foo << results[i] << endl;
}
papiOut.close();

#else // DOTIME or DOPROF
doIt();
#endif
};

////////////////////////////////////

```

Monday December 02, 2002

Nov 30, 02 21:19

main.h

Page 4/4

```

void startMeasure() {
    // Write pid to file. This pid is used for the benchmark driver to
    // kill processes that took too long.

    ofstream pidOut;
    pidOut.open("pid.dat");
    pidOut << getpid() << endl;
    pidOut.close();

#ifdef DOPAPI
printf("Starting papi, LENGTH: %d\n", LENGTH);
int err = PAPI_start_counters(&events[done], todo);
if (err != 0) {
    printf("PAPI_start failed with error %d, done: %d\n", err, done);
    PAPI_perror(err, 0, 0);
    exit(-2);
}
#endif
#ifdef DOTIME
// Start timer.
gettimeofday(&tv, 0);
beginMS = tv.tv_usec;
beginS = tv.tv_sec;
#endif
#ifdef DOPROC
startReadPageFaults();
#endif
// Doing nothing when DOTRACE
}

////////////////////////////////////
void stopMeasure()
{
#ifdef DOPAPI
int err = PAPI_stop_counters(&results[done], todo);
if (err != 0) {
    cerr << " PAPI_stop failed with code: " << err << endl;
    PAPI_perror(err, 0, 0);
    exit(-2);
}
#endif
#ifdef DOTIME
// Stop timer.
gettimeofday(&tv, 0);
endMS = tv.tv_usec;
endS = tv.tv_sec;
// printf("endtime: %f:%f\n", endS, endMS);

// Print result to stdout and to file.
double result = (endS - beginS) * 1000000 + (endMS - beginMS);
printf("Time to compute: %f\n", result);

ofstream timeOut;
timeOut.open("timeit.dat");
timeOut << result;
timeOut.close();
#endif
#ifdef DOPROC
stopReadPageFaults();
#endif
}

#endif // SYSTEM_MAIN

```

src/template/main.h

35/36

Oct 28, 02 15:42

out_of_store.h

Page 1/1

```
////////////////////////////////////  
//  
// out_of_store.h  
//  
// By executing the command: set_new_handler(out_of_store); and  
// including this file, new will write an error to the user, when  
// memory is exhausted.  
////////////////////////////////////  
  
#ifndef OUTOFSTORE  
#define OUTOFSTORE  
  
#include <exception>  
  
void out_of_store() {  
    cerr << "Memory Exhausted. There is not room for all elements in " <<  
        "the memory-space.\n";  
    throw std::bad_alloc();  
    //exit(-1);  
}  
  
#endif
```