Heap ADT

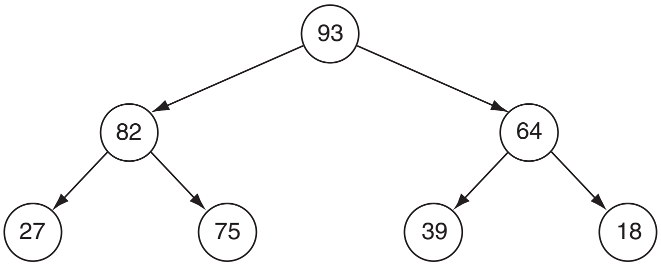
objectives

In this laboratory you will:

- Create an implementation of the Heap ADT using an array representation of a tree
- Use inheritance to derive a priority queue class from your heap class and develop a simulation of an operating system's task scheduler using a priority queue
- Create a heap sort function based on the heap construction techniques used in your implementation of the Heap ADT
- Analyze where data items with various priorities are located in a heap

Overview

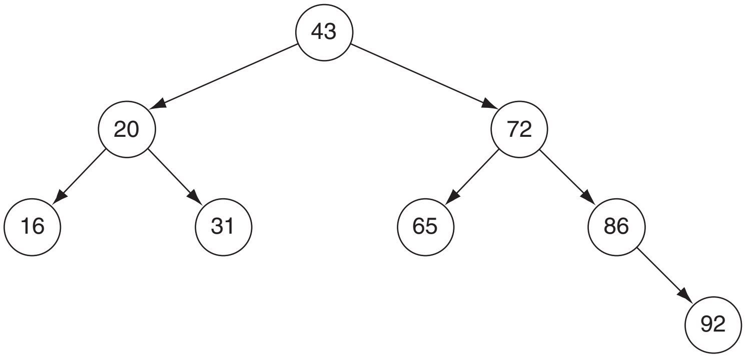
Linked structures are not the only way in which you can represent trees. If you take the binary tree shown below and copy its contents into an array in level order, you produce the following array.



Index	Entry
0	93
1	82
2	64
3	27
4	75
5	39
6	18

Examining the relationship between positions in the tree and entries in the array, you see that if a data item is stored in entry N in the array, then the data item's left child is stored in entry $2N + 1$, its right child is stored in entry $2N + 2$, and its parent is stored in entry $(N - 1)$ mod 2. These mappings make it easy to move through the tree stepping from parent to child (or vice versa).

You could use these mappings to support an array-based implementation of the Binary Search Tree ADT. However, the result would be a tree representation in which large areas of the array are left unused (as indicated by the "–" character in the following array).

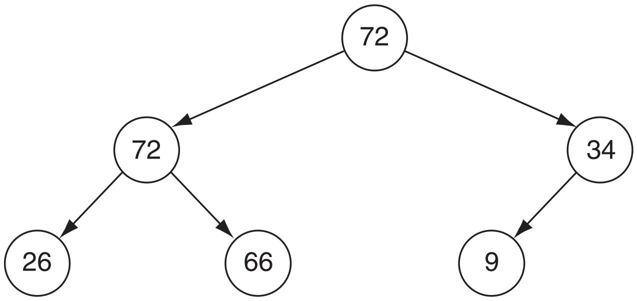


Index	Entry
0	43
1	20
2	72
3	16
4	31
5	65
6	86
7	–
8	–
...	...
14	–
15	92

In this laboratory, you focus on a different type of tree called a heap. A **heap** is a binary tree that meets the following conditions.

- The tree is **complete**. That is, every level in the tree is full, except possibly the bottom level. If the bottom level is not full, then all the missing data items occur on the right.
- Each data item in the tree has a corresponding priority value. For each data item E, all of E's descendants have priorities that are less than or equal to E's priority. Note that priorities are *not* unique.

The tree shown on the first page of this laboratory is a heap, as is the tree shown below.



The fact that the tree is complete means that a heap can be stored in level order in an array without introducing gaps (unused areas) in the middle. The result is a compact representation in which you can easily move up and down the branches.

Clearly, the relationship between the priorities of the various data items in a heap is not strong enough to support an efficient search process. Because the relationship is simple, however, you can quickly restructure a heap after removing the highest priority (root) data item or after inserting a new data item. As a result, you can rapidly process the data items in a heap in descending order based on priority. This property combined with the compact array representation makes a heap an ideal representation for a priority queue (In-lab Exercise 1) and forms the basis for an efficient sorting algorithm called heap sort (In-lab Exercise 2).

Heap ADT

Data Items

The data items in a heap are of generic type DT. Each data item has a priority that is used to determine the relative position of the data item within the heap. Data items usually include additional data. Note that priorities are *not* unique—it is quite likely that several data items have the same priority. Objects of type DT must provide a function called `pty()` that returns a data item's priority. You must be able to compare priorities using the six basic relational operators.

Structure

The data items form a complete binary tree. For each data item *E* in the tree, all of *E*'s descendants have priorities that are less than or equal to *E*'s priority.

Operations

`Heap ( int maxNumber = defMaxHeapSize )`

Requirements:
None

Results:
Constructor. Creates an empty heap. Allocates enough memory for a heap containing `maxNumber` data items.

```
~Heap ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store a heap.

```
void insert ( const DT &newDataItem ) throw ( logic_error )
```

Requirements:
Heap is not full.

Results:
Inserts `newDataItem` into a heap. Inserts this data item as the bottom rightmost data item in the heap and moves it upward until the properties that define a heap are restored.

```
DT removeMax () throw ( logic_error )
```

Requirements:
Heap is not empty.

Results:
Removes the data item with the highest priority (the root) from a heap and returns it. Replaces the root data item with the bottom rightmost data item and moves this data item downward until the properties that define a heap are restored.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in a heap.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if a heap is empty. Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a heap is full. Otherwise, returns `false`.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the priorities of the data items in a heap in both array and tree form. The tree is output with its branches oriented from left (root) to right (leaves)—that is, the tree is output rotated counterclockwise 90 degrees from its conventional orientation. If the heap is empty, outputs "Empty heap". Note that this operation is intended for testing/debugging purposes only.

Laboratory B: Cover Sheet

Name ______________________________ Date ____________________

Section ______________________________

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory B: Prelab Exercise

Name ____ Date ________________________

Section ____

Step 1: Implement the operations in the Heap ADT using an array representation of a heap. Heaps can be different sizes; therefore, you need to store the maximum number of data items the heap can hold (`maxSize`) and the actual number of data items in the heap (`size`), along with the heap data items themselves (`dataItems`). Base your implementation on the following declarations from the file *heap.h*. An implementation of the `showStructure` operation is given in the file *showb.cpp*.

```
const int defMaxHeapSize = 10;    // Default maximum heap size

template < class DT >
class Heap
{
  public:

    // Constructor
    Heap ( int maxNumber = defMaxHeapSize ) throw ( bad_alloc );

    // Destructor
    ~Heap ();

    // Heap manipulation operations
    void insert ( const DT &newDataItem )     // Insert data item
        throw ( logic_error );
    DT removeMax () throw ( logic_error );    // Remove max pty data item
    void clear ();                            // Clear heap

    // Heap status operations
    int isEmpty () const;                     // Heap is empty
    int isFull () const;                      // Heap is full

    // Output the heap structure — used in testing/debugging
    void showStructure () const;

  private:

    // Recursive partner of the showStructure() function
    void showSubtree ( int index, int level ) const;

    // Data members
    int maxSize,                         // Maximum number of data items in the
heap
        size;                            // Actual number of data items in the heap
    DT *dataItems;                       // Array containing the heap data items
};
```

Step 2: Save your implementation of the Heap ADT in the file *heap.cpp*. Be sure to document your code.

Laboratory B: Bridge Exercise

Name ______________________________ Date ____________________

Section ______________________________

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

The test program in the file *testb.cpp* allows you to interactively test your implementation of the Heap ADT using the following commands.

Command	Action
`+pty`	Insert a data item with the specified priority.
`-`	Remove the data item with the highest priority from the heap and output it.
`E`	Report whether the heap is empty.
`F`	Report whether the heap is full.
`C`	Clear the heap.
`Q`	Quit the test program.

Step 1: Prepare a test plan for your implementation of the Heap ADT. Your test plan should cover heaps of various sizes, including empty, full, and single data item heaps. A test plan form follows.

Step 2: Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.

Test Plan for the Operations in the Heap ADT

Test Case	*Commands*	*Expected Result*	*Checked*

Laboratory B: In-lab Exercise 1

Name ______________________________ Date ____________________

Section ______________________________

A **priority queue** is a linear data structure in which the data items are maintained in descending order based on priority. You can only access the data item at the front of the queue—that is, the data item with the highest priority—and examining this data item entails removing (dequeuing) it from the queue.

Priority Queue ADT

Data Items

The data items in a priority queue are of generic type DT. Each data item has a priority that is used to determine the relative position of the data item within the queue. Data items usually include additional data. Objects of type DT must supply a function called `pty()` that returns a data item's priority. You must be able to compare priorities using the six basic relational operators.

Structure

The queue data items are stored in descending order based on priority.

Operations

```
Queue ( int maxNumber = defMaxQueueSize )
```

Requirements:
None

Results:
Constructor. Creates an empty priority queue. Allocates enough memory for a queue containing `maxNumber` data items.

```
~Queue ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store a priority queue.

```
void enqueue ( const DT &newDataItem ) throw ( logic_error )
```

Requirements:
Queue is not full.

Results:
Inserts `newDataItem` into a priority queue.

```
DT dequeue () throw ( logic_error )
```

Requirements:
Queue is not empty.

Results:
Removes the highest priority (front) data item from a priority queue and returns it.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in a priority queue.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if a priority queue is empty. Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a priority queue is full. Otherwise, returns `false`.

You can easily and efficiently implement a priority queue as a heap by using the Heap ADT `insert` operation to enqueue data items and the `removeMax` operation to dequeue data items. The following declarations from the file *ptyqueue.h* derive a class called PtyQueue from the Heap class. If you are unfamiliar with the C++ inheritance mechanism, read the discussion in Laboratory 4.

```
const int defMaxQueueSize = 10;   // Default maximum queue size

template < class DT >
class PtyQueue : public Heap<DT>
{
  public:

    // Constructor
    PtyQueue ( int maxNumber = defMaxQueueSize );

    // Queue manipulation operations
    void enqueue ( const DT &newDataItem ) throw ( logic_error );
                                                      // Enqueue data data item
    DT dequeue () throw ( logic_error );
                                                      // Dequeue data data item
};
```

Implementations of the Priority Queue ADT `constructor`, `enqueue`, and `dequeue` operations are given in the file *ptyqueue.cpp*. These implementations are very short, reflecting the close relationship between the Heap ADT and the Priority Queue ADT. Note that you inherit the remaining operations in the Priority Queue ADT from the Heap class.

Operating systems commonly use priority queues to regulate access to system resources such as printers, memory, disks, software, and so forth. Each time a task requests access to a system resource, the task is placed on the priority queue associated with that resource. When the task is dequeued, it is granted access to the resource—to print, store data, and so on.

Suppose you wish to model the flow of tasks through a priority queue having the following properties:

- One task is dequeued every minute (assuming that there is at least one task waiting to be dequeued during that minute).
- From zero to two tasks are enqueued every minute, where there is a 50% chance that no tasks are enqueued, a 25% percent chance that one task is enqueued, and a 25% chance that two tasks are enqueued.
- Each task has a priority value of zero (low) or one (high), where there is an equal chance of a task having either of these values.

You can simulate the flow of tasks through the queue during a time period *n* minutes long using the following algorithm.

Initialize the queue to empty.
for (minute = 0 ; minute < n ; ++minute)
{
  If the queue is not empty, then remove the task at the front of the queue.
  Compute a random integer *k* between 0 and 3.
  If *k* is 1, then add one task to the queue. If *k* is 2, then add two tasks.
    Otherwise (if *k* is 0 or 3), do not add any tasks to the queue. Compute the
    priority of each task by generating a random value of 0 or 1.
}

Step 1: Using the program shell given in the file *ossim.cs* as a basis, create a program that uses the Priority Queue ADT to implement the task scheduler described above. Your program should output the following information about each task as it is dequeued: the task's priority, when it was enqueued, and how long it waited in the queue.

Step 2: Use your program to simulate the flow of tasks through the priority queue and complete the following table.

Time (minutes)	*Longest wait for any low-priority (0) task*	*Longest wait for any high-priority (1) task*
10		
30		
60		

Step 3: Is your priority queue task scheduler unfair—that is, given two tasks T_1 and T_2 of the same priority, where task T_1 is enqueued at time N and task T_2 is enqueued at time $N + i$ $(i > 0)$, is task T_2 ever dequeued before task T_1? If so, how can you eliminate this problem and make your task scheduler fair?

Laboratory B: In-lab Exercise 2

Name ______________________________ Date ____________________

Section ______________________________

After removing the root data item, the `removeMax` operation inserts a new data item at the root and moves this data item downward until a heap is produced. The following function performs a similar task, except that the heap it is building is rooted at array entry root and occupies only a portion of the array.

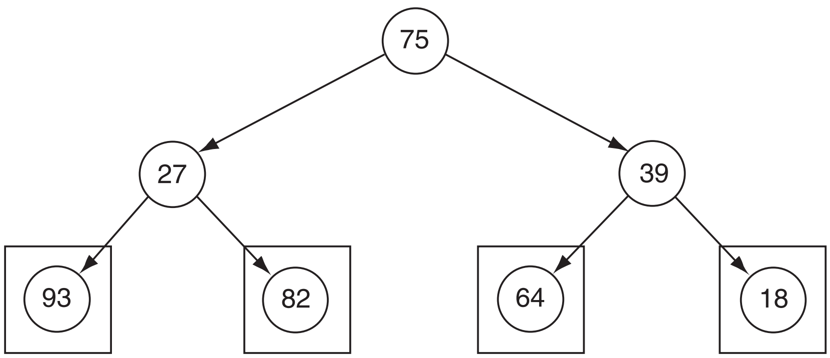
```
void moveDown ( DT dataItems [], int root, int size )
```

Input:
The left and right subtrees of the binary tree rooted at `root` are heaps. Parameter `size` is the number of elements in the tree.

Output:
Restores the binary tree rooted at `root` to a heap by moving `dataItems[root]` downward until the tree satisfies the heap property.

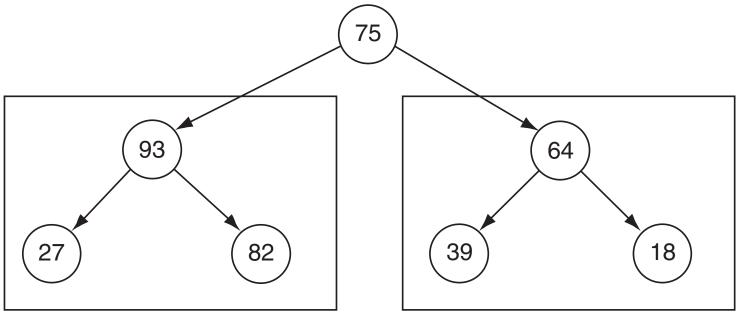
In this exercise, you implement an efficient sorting algorithm called **heap sort** using the `moveDown()` function. You first use this function to transform an array into a heap. You then remove data items one by one from the heap (from the highest priority data item to the lowest) until you produce a sorted array.

Let's begin by examining how you transform an unsorted array into a heap. Each leaf of any binary tree is a one-data item heap. You can build a heap containing three data items from a pair of sibling leaves by applying the `moveDown()` function to that pair's parent. The four single data item heaps (leaf nodes) in the following tree are transformed by the calls `moveDown(dataItems,1,7)` and `moveDown(dataItems,2,7)` into a pair of three data item heaps.



Index	Entry
0	75
1	27
2	39
3	93
4	82
5	64
6	18

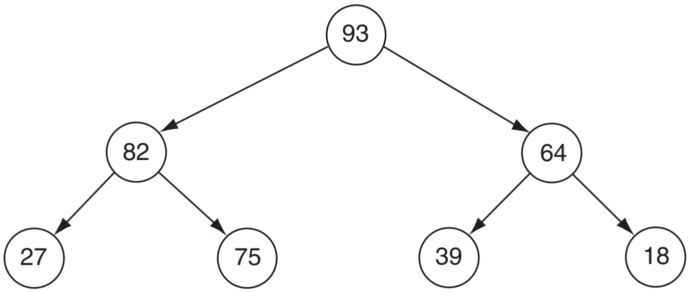
By repeating this process, you build larger and larger heaps, until you transform the entire tree (array) into a heap.



Index	Entry
0	75
1	93
2	64
3	27
4	82
5	39
6	18

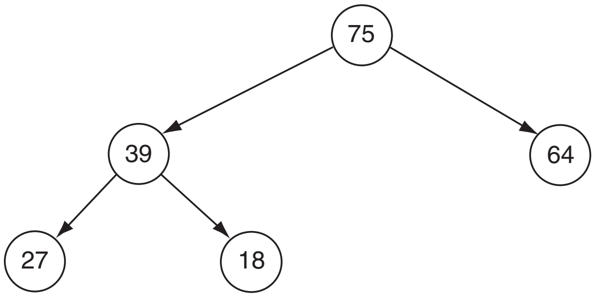
```
// Build successively larger heaps within the array until the
// entire array is a heap.

for ( j = (size-1)/2 ; j >= 0 ; j-- )
      moveDown(dataItems,j,size);
```

Combining the pair of three-data item heaps shown above using the call `moveDown(dataItems,0,7)`, for instance, produces the following heap.



Index	Entry
0	93
1	82
2	64
3	27
4	75
5	39
6	18

Now that you have a heap, you remove data items of decreasing priority from the heap and gradually construct an array that is sorted in ascending order. The root of the heap contains the highest-priority data item. If you swap the root with the data item at the end of the array and use `moveDown()` to form a new heap, you end up with a heap containing six data items and a sorted array containing one data item. Performing this process a second time yields a heap containing five data items and a sorted array containing two data items.



	Index	Entry
Heap	0	75
	1	39
	2	64
	3	27
	4	18
Sorted array	5	82
	6	93

You repeat this process until the heap is gone and a sorted array remains.

```
// Swap the root data item from each successively smaller heap with
// the last unsorted data item in the array. Restore the heap after
// each exchange.

for ( j = size-1 ; j > 0 ; j-- )
{
    temp = dataItems[j];
    dataItems[j] = dataItems[0];
    dataItems[0] = temp;
    moveDown(dataItems,0,j);
}
```

A shell containing a `heapSort()` function comprised of the two loops shown above is given in the file *heapsort.cs*.

Step 1: Using your implementation of the `removeMax` operation as a basis, create an implementation of the `moveDown()` function.

Step 2: Add your implementation of the `movedown()` function to the shell in the file *heapsort.cs* thereby completing code needed by the `heapSort()` function. Save the result in the file *heapsort.cpp*.

Step 3: Before testing the resulting `heapSort()` function using the test program in the file *testbhs.cpp*, prepare a test plan for the `heapSort()` function that covers arrays of different lengths containing a variety of priority values. Be sure to include arrays that have multiple data items with the same priority. A test plan form follows.

Step 4: Execute your test plan. If you discover mistakes in your implementation of the `moveDown()` function, correct them and execute your test plan again.

Test Plan for the `heapSort` Operation

Test Case	*Array*	*Expected Result*	*Checked*