Chapter 12
Heaps

• Introduction to Heaps
• The Heap Workshop Applet
• Java Code for Heaps
• A Tree-based Heap
• Heapsort
• Summary

Overview

• Chapter 4, “Stacks and Queues,”
  – a priority queue is a data structure that offers convenient access to the data item with the smallest (or largest) key.
  – This is useful when key values indicate the order in which items should be accessed.
• Priority queues may be used for task scheduling in computers, where some programs and activities should be executed sooner than others and are therefore given a higher priority.
• Another example is in weapons systems, say in a navy cruiser.
  – A variety of threats—airplanes, missiles, submarines, and so on—are detected and must be prioritized.
  – For example, a missile that’s a short distance from the cruiser is assigned a higher priority than an aircraft a long distance away, so that countermeasures (surface-to-air missiles, for example) can deal with it first.

Overview

• Priority queues are also used internally in other computer algorithms.
  – In Chapter 14, “Weighted Graphs,”
    • priority queues used in graph algorithms, such as Dijkstra’s Algorithm.
• A priority queue is an Abstract Data Type (ADT) offering methods that allow removal of the item with the maximum (or minimum) key value, insertion, and sometimes other activities.
• As with other ADTs, priority queues can be implemented using a variety of underlying structures.
  – In Chapter 4 we saw a priority queue implemented as an array.
    • The trouble with that approach is that, even though removal of the largest item is accomplished in fast $O(1)$ time, insertion requires slow $O(N)$ time, because an average of half the items in the array must be moved to insert the new one in order.

Overview

• Another structure that can be used to implement a priority queue: the heap.
• A heap is a kind of tree.
  – It offers both insertion and deletion in $O(\log N)$ time.
  – Thus it’s not quite as fast as deletion, but much faster for insertion.
  – It’s the method of choice for implementing priority queues where speed is important and there will be many insertions.
  – (Incidentally, don’t confuse the term heap, used here for a special kind of binary tree, with the same term used to mean the portion of computer memory available to a programmer with new in languages like Java and C++.)

Introduction to Heaps

Review: CBT and BST

• CBT (Complete Binary Tree)
  – Given the node with index $i$
    • parent’s index is $(i-1)/2$
    • left child’s index is $(2i+1)$
    • right child’s index is $(2i+2)$
• BST (Binary Search Tree)
  – For each node
    • the key of every node in the left subtree is smaller than the key of this node
    • the key of every node in the right subtree is larger than the key of this node

Introduction to Heaps

• Heap is
  – Complete Binary Tree
  – For any node $n$, both children’s keys are less than its key
Introduction to Heaps

Heap To Array

• Similar to CBT, we can easily get an array from Heap
  – Get the elements from top to bottom, from left to right
  – Array: \{ 27, 23, 17, 14, 19, 6, 10, 7, 3 \}

Introduction to Heaps

Array Representation For Heap

• Use array to represent the heap
  – Simpler than using tree to present the heap
• Two properties for any index i in array
  – array[i] > array[2*i+1]
  – array[i] > array[2*i+2]
  – here 2*i+2 is less than array.length
• Example
  – 27, 23, 17, 14, 19, 6, 10, 7, 3

Introduction to Heaps

Add New Item In Heap

• Given a heap
• Add one more item at end of heap
• Is there any easy way to make this new tree still to be a heap?

Introduction to Heaps

Add Item In Heap

• Add new node 35 in this heap
• step 1: add new node in the leftmost available slot on the deepest level

Introduction to Heaps

Add Item In Heap (Cont.)

• step 2: swap the child and parent if it violates the heap property

Introduction to Heaps

Add Item In Heap (Cont.)

• step 3: repeat swap operations if it still violates the heap property
**Introduction to Heaps**

**Cost For Add Operation**

- How many swaps are needed for add operation?
  - At most $\text{height} - 1$
- For $N$ nodes Heap,
  - Height is $\lceil \log_2(N + 1) \rceil$
  - So, cost for Add operation:
    - $O(\log N)$

**Delete Item In Heap**

- delete node 27
  - the same as deleting root in subtree below

**Delete Root In Heap**

- So, we focus on deleting root in heap
- Delete node 35 from heap

**Delete Root In Heap (Cont.)**

- Step 1: put the rightmost node in the deepest level into the root

**Delete Root In Heap (Cont.)**

- Step 2: swap the child and parent if it violates the heap property

**Delete Root In Heap (Cont.)**

- Step 3: repeat swap operations if it still violates the heap property
Introduction to Heaps
Cost For Delete Operation

• How many swaps are needed for delete operation?
  – At most \textit{height} – 1
• For N nodes Heap,
  – Height is \[
    \left\lceil \log_2(N + 1) \right\rceil
  \]
  – So, cost for Delete operation:
    • \(O(\log N)\)

Introduction to Heaps
Return Maximum Node From Heap

• Root is the maximum node
• Cost: \(O(1)\)

Introduction to Heaps
Implement Heap As Array

• It is very easy to use array to implement heap
  – \texttt{array[i]’s parent is array[(i - 1) / 2]}
  – \texttt{array[i]’s left child is array[2*i+1]}
  – \texttt{array[i]’s right child is array[2*i+2]}
• Easy to swap child and parent

Introduction to Heaps
Heapify An Array

• Make an unsorted array as heap

\[
\begin{array}{c}
8 & 20 & 33 & 10 & 7 & \ldots & 15 & 19 & 27 \\
\end{array}
\]

• Consider the first item, it is one-item heap

\[
\begin{array}{c}
8 & 20 & 33 & 10 & 7 & \ldots & 15 & 19 & 27 \\
\end{array}
\]

Introduction to Heaps
Heapify An Array (Cont)

• Add new item to heap

\[
\begin{array}{c}
8 & 33 & 10 & 7 & \ldots & 15 & 19 & 27 \\
\end{array}
\]

• Add next new item to heap

\[
\begin{array}{c}
20 & 8 & 33 & 10 & 7 & \ldots & 15 & 19 & 27 \\
\end{array}
\]

• Repeat add operation until to last element
  – call \texttt{add item in heap}, so the array is:

\[
\begin{array}{c}
33 & 8 & 20 & 10 & 7 & \ldots & 15 & 19 & 27 \\
\end{array}
\]

• At last the array has heap property
The Heap Workshop Applet

- It allows you to insert new items into a heap and remove the largest item.
- In addition you can change the priority of a given item.
- When you start up the Heap Workshop applet, you’ll see a display similar to Figure 12.7.
- There are four buttons: Fill, Chng, Rem, and Ins, for fill, change, remove, and insert. Let’s see how they work.

Fill
- The heap contains 10 nodes when the applet is first started.
- Using the Fill key you can create a new heap with any number of nodes from 1 to 31.
  - Press Fill repeatedly, and type in the desired number when prompted.

Change
- It’s possible to change the priority of an existing node.
  - This is a useful procedure in many situations.
    - For example, in our cruiser example, a threat such as an approaching airplane may reverse course away from the carrier; its priority should be lowered to reflect this new development, although the aircraft would remain in the priority queue until it was out of radar range.
  - To change the priority of a node, repeatedly press the Chng key.
    - When prompted, click on the node with the mouse.
      - This will position the red arrow on the node.
    - Then, when prompted, type in the node’s new priority.
    - If the node’s priority is raised, it will trickle upward to a new position.
    - If the priority is lowered, the node will trickle downward.

Remove
- Repeatedly pressing the Rem button causes the node with the highest key, located at the root, to be removed.
- You’ll see it disappear, and then be replaced by the last (rightmost) node on the bottom row.
- Finally this node will trickle down until it reaches the position that reestablishes the heap order.

Insert
- A new node is always inserted initially in the first available array cell, just to the right of the last node on the bottom row of the heap.
  - From there it trickles up to the appropriate position.
  - Pressing the Ins key repeatedly carries out this operation.

Java Code for Heaps
- The complete code for heap.java is shown later in this section.
- For a node at index x in the array,
  - Its parent is \((x-1)/2\)
  - Its left child is \(2x + 1\)
  - Its right child is \(2x + 2\)
- (The / symbol, when applied to integers, performs integer division, in which the answer is rounded to the lowest integer.)
Java Code for Heaps

Insertion
• We place the trickle-up algorithm in its own method.
• The `insert()` method, which includes a call to this `trickleUp()` method, is straightforward:
  ```java
  public boolean insert(int key)
  {
    if (currentSize == maxSize) // if array is full,
      return false; // failure
    Node newNode = new Node(key); // make a new node
    heapArray[currentSize++] = newNode; // put it at the end
    trickleUp(currentSize++); // trickle it up
    return true; // success
  }
  ```
• We check to make sure the array isn’t full and then make a new node
  using the key value passed as an argument.
• Finally the `trickleUp()` routine is called to move this node up to its
  proper position.

Java Code for Heaps

Removal
• The `remove()` routine is more complicated than `trickleUp()` because we must determine which of
  the two children is larger.
  - First we save the node at index in a variable called `top`.
  - If `trickleDown()` has been called from `remove()`, index is
    the root;
    • but, as we’ll see, it can be called from other routines as well.
  - The while loop will run as long as index is not on the
    bottom row—that is, as long as it has at least one child.
  - Within the loop we check if there is a right child (there may be
    only a left):
    • if so, compare the children’s keys, setting `largerChild`
      appropriately.
  - Then we check if the key of the original node (now in top) is
    greater than that of `largerChild`:
    • if so, the trickle-down process is complete and we exit the loop.
  ```java
  public Node remove() // delete item with max key
  { // (assumes non-empty list)
    Node root = heapArray[0]; // save the root
    heapArray[0] = heapArray[--currentSize]; // root <- last
    trickleDown(0); // trickle down the root
    return root; // return removed node
  }
  ```
• This method returns the node that was removed; the
  user of the heap usually needs to process it in some
  way.

Java Code for Heaps

Removal
• The removal algorithm is also not complicated if we
  subsume the trickle-down algorithm into its own routine.
• We save the node from the root, copy the last node (at
  index `currentSize-1`) into the root, and call
  `trickleDown()` to place this node in its appropriate
  location.
  ```java
  public Node remove() // delete item with max key
  { // (assumes non-empty list)
    Node root = heapArray[0]; // save the root
    heapArray[0] = heapArray[--currentSize]; // root <- last
    trickleDown(0); // trickle down the root
    return root; // return removed node
  }
  ```
• This method returns the node that was removed; the
  user of the heap usually needs to process it in some
  way.

Java Code for Heaps

Insertion
• In `trickleUp()` (shown in next slide) the argument is the index of
  the newly inserted item.
• We find the parent of this position and then save the node in a
  variable called `bottom`.
• Inside the while loop, the variable `index` will trickle up the path
  toward the root, pointing to each node in turn.
  • (This has the effect of moving the “hole” upward.)
  • Then it moves index upward by giving it its parent’s index, and giving its
    parent its parent’s index.
  ```java
  public void trickleUp(int index)
  {
    int parent = (index-1) / 2;
    Node bottom = heapArray[index];
    while( index > 0 && heapArray[parent].iData < bottom.iData )
    {
      heapArray[index] = heapArray[parent]; // move node down
      index = parent; // move index up
      parent = (parent-1) / 2; // parent <- its parent
    }
    heapArray[index] = bottom;
  }
  ```
• On exiting the loop we need only restore the node stored in top to its appropriate
  position, pointed to by index.
The user displays the heap, adds an item with a key of 53, shows the heap again, removes the item with the greatest key, shows the heap again, and then displays the heap a third time. The last time the heap is displayed, the user enters s, i, r, or c, for show, insert, remove, or change.

Here's some sample interaction with the program:

The Array Size

- We should note that the array size, equivalent to the number of nodes in the heap, is a vital piece of information about the heap's state and a critical field in the Heap class.
- Nodes copied from the last position aren't erased, so the only way for algorithms to know the location of the last occupied cell is to refer to the current size of the array.

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```

Java Code for Heaps

```java
private boolean isEmpty()
{
    return true;
}
```

Java Code for Heaps

```java
public void trickleDown(int index)
{
    if(index<0 || index>=currentSize)
    { // end trickleDown()}
}
```

Java Code for Heaps

```java
public void trickleUp(int index)
{
    if(oldValue < newValue) // if raised,
    { // end trickleUp()}
}
```

Java Code for Heaps

```java
public boolean change(int index, int newValue)
{
    int oldValue = heapArray[index].iData; // remember old
    heapArray[index].iData = newValue; // change to new
    if(index<0 || index>=currentSize)
    { // end change()}
}
```
Java Code for Heaps

**Efficiency of Heap Operations**

- For a heap with a substantial number of items, it's the `trickleUp` and `trickleDown` algorithms that are the most time-consuming parts of the operations we've seen.
  - These algorithms spend time in a loop, repeatedly moving nodes up or down along a path.
  - The number of copies necessary is bounded by the height of the heap;
    - if there are five levels, four copies will carry the "hole" from the top to the bottom.
    - (We'll ignore the two moves used to transfer the end node to and from temporary storage; they're always necessary so they require constant time.)

- The `trickleUp()` method has only one major operation in its loop: comparing the key of the new node with the node at the current location.
- The `trickleDown()` method needs two comparisons:
  - one to find the largest child.
    - They must both copy a node from top to bottom or bottom to top to complete the operation.
  - A heap is a special kind of binary tree,
    - as we saw in Chapter 8, the number of levels \( L \) in a binary tree equals \( \log_2(N+1) \), where \( N \) is the number of nodes.
    - The `trickleUp()` and `trickleDown()` routines cycle through their loops \( L-1 \) times, so the first takes time proportional to \( \log_2 N \), and the second somewhat more because of the extra comparison.
    - Thus the heap operations we've talked about here all operate in \( O(\log N) \) time.

Heapsort

**Efficiency of Heap Operations**

- The efficiency of the heap data structure lends itself to a surprisingly simple and very efficient sorting algorithm called heapsort.
- The basic idea is to insert the unordered items into a heap using the normal `insert()` routine.
- Repeated application of the `remove()` routine will then remove the items in sorted order.
- Here's how that might look:
  ```java
  for(j=0; j<size; j++)
  theHeap.insert( anArray[j] ); // from unsorted array
  for(j=0; j<size; j++)
  anArray[j] = theHeap.remove(); // to sorted array
  ```
- Because `insert()` and `remove()` operate in \( O(\log N) \) time, and each must be applied \( N \) times, the entire sort requires \( O(N \log N) \) time, which is the same as quicksort.
  - However, it's not quite as fast as quicksort.
    - Partly this is because there are more operations in the inner `while` loop in `trickleDown()` than in quicksort.
  - However, several tricks can make heapsort more efficient.
    - The first saves time, and the second saves memory.

Trickling Down in Place

**Two Correct Subheaps Make a Correct Heap**

- As we know, `trickleDown()` will create a correct heap if, when an out-of-order item is placed at the root, both the child subheaps of this root are correct heaps.
- (The root can itself be the root of a subheap as well as of the entire heap.)
- This is shown in Figure 12.8.
- We can apply `trickleDown()` to the nodes on the bottom of the (potential) heap
  - note that the end of the array
  - and work our way upward to the root at index 0.
  - At each step the subheaps below us will already be correct heaps because we already applied `trickleDown()` to them.
  - After we apply `trickleDown()` to the root, the unordered array will have been transformed into a heap.

Heapsort

**Trickling Down in Place**

- If we insert \( N \) new items into a heap, we apply the `trickleUp()` method \( N \) times.
- However, if all the items are already in an array, they can be rearranged into a heap with only \( N/2 \) applications of `trickleDown()`.
  - This offers a small speed advantage.

Heapsort

**Trickling Down in Place**

- Notice
  - the nodes on the bottom row—those with no children—are already correct heaps, because they are trees with only one node;
  - they have no relationships to be out of order.
- We can start at node \( N/2-1 \), the rightmost node with children, instead of \( N-1 \), the last node.
- Thus we need only half as many trickle operations as we would using `insert()` \( N \) times.
- Figure 12.9 shows the order in which the trickle-down algorithm is applied, starting at node 6 in a 15-node heap.
- The following code fragment applies `trickleDown()` to all nodes, except those on the bottom row, starting at \( N/2-1 \) and working back to the root:
  ```java
  for(j=size/2-1; j >=0; j--)
  theHeap.trickleDown(j);
  ```

Heapsort

**Two Correct Subheaps Make a Correct Heap**

- The `trickleUp()` method has only one major operation in its loop: comparing the key of the new node with the node at the current location.
- The `trickleDown()` method needs two comparisons,
  - one to find the largest child.
    - They must both copy a node from top to bottom or bottom to top to complete the operation.
  - A heap is a special kind of binary tree,
    - as we saw in Chapter 8, the number of levels \( L \) in a binary tree equals \( \log_2(N+1) \), where \( N \) is the number of nodes.
    - The `trickleUp()` and `trickleDown()` routines cycle through their loops \( L-1 \) times, so the first takes time proportional to \( \log_2 N \), and the second somewhat more because of the extra comparison.
    - Thus the heap operations we've talked about here all operate in \( O(\log N) \) time.
Heapsort

Trickling Down in Place

A Recursive Approach
- A recursive approach can also be used to form a heap from an array
  - `heapify()` method is applied to the root.
    - It calls itself for the root’s two children, then for each of these children’s two children, and so on.
    - Eventually it works its way down to the bottom row, where it returns immediately whenever it finds a node with no children.
- Once it has called itself for two child subtrees, `heapify()` then applies `trickleDown()` to the root of the subtree.
  - This ensures that the subtree is a correct heap.
- Then `heapify()` returns and works on the subtree one level higher.

```java
heapify(int index) // transform array into heap
if(index > N/2-1) // if node has no children,
    return; // return
heapify(index*2+1); // turn left subtree into heap
heapify(index*2+2); // turn right subtree into heap
trickleDown(index); // apply trickle-down to this node
```
- This recursive approach is probably not quite as efficient as the simple loop.

Heapsort

Using the Same Array

- However, things are more complicated when we apply `remove()` repeatedly to the heap.
  - Where are we going to put the items that are removed?
- Each time an item is removed from the heap, an element at the end of the heap array becomes empty; the heap shrinks by one.
- We can put the recently removed item in this newly freed cell.
- As more items are removed, the heap array becomes smaller and smaller, while the array of ordered data becomes larger and larger.
  - Thus with a little planning it’s possible for the ordered array and the heap array to share the same space.
  - This is shown in Figure 12.10.

Heapsort

The `heapSort().java` Program

- We can put these two tricks together in a program that performs heapsort.
- Increment `size` as appropriate, rather than removing the last element.
- In this procedure two size-N arrays are required: the initial array and the heap,
  - Where are we going to put the items that are removed?

- This data was then inserted into a heap, and finally removed from the heap and written back to the array in sorted order.
- This addition is not in the spirit of object-oriented programming.
  - The heap’s data interface is exposed to class clients.
  - The underlying array should be invisible, but `incrementSize()` allows direct access to it.
- In this situation we accept the violation of OOP principles because the array is so closely tied to the heap.
- An `incrementSize()` method is another addition to the heap class.
  - It might seem as though we could combine this with `incrementSize()`, but when inserting into the heap in its role as an ordered array we don’t want to increase the heap size, so we keep these functions separate.

Heapsort

The Efficiency of Heapsort

- As we noted, heapsort runs in $O(N \log N)$ time.
- Although it may be slightly slower than quicksort,
  - an advantage over quicksort is that it is less sensitive to the initial distribution of data.
- Certain arrangements of key values can reduce quicksort to slow $O(N^2)$ time, whereas heapsort runs in $O(N \log N)$ time no matter how the data is distributed.
The Efficiency of Heapsort

- Cost: $O(N \log N)$
- Comparison to other sorting algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Time Cost</th>
<th>Space Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$O(N^2)$</td>
<td>1</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(N^2)$</td>
<td>1</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(N \log N)$</td>
<td>$N$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(N \log N)$</td>
<td>1</td>
</tr>
</tbody>
</table>

Summary (I)

- In an ascending priority queue the item with the largest key (or smallest in a descending queue) is said to have the highest priority.
- A priority queue is an Abstract Data Type (ADT) that offers methods for insertion of data and removal of the largest (or smallest) item.
- A heap is an efficient implementation of an ADT priority queue.
- A heap offers removal of the largest item, and insertion, in $O(N \log N)$ time.
- The largest item is always in the root.
- Heaps do not support ordered traversal of the data, locating an item with a specific key, or deletion.

Summary (II)

- A heap is usually implemented as an array representing a complete binary tree. The root is at index 0 and the last item at index $N-1$.
- Each node has a key less than its parents and greater than its children.
- An item to be inserted is always placed in the first vacant cell of the array, and then trickled up to its appropriate position.
- When an item is removed from the root, it’s replaced by the last item in the array, which is then trickled down to its appropriate position.
- The trickle-up and trickle-down processes can be thought of as a sequence of swaps, but are more efficiently implemented as a sequence of copies.

Summary (III)

- The priority of an arbitrary item can be changed. First its key is changed; then, if the key was increased, the item is trickled up, while if the key was decreased the item is trickled down.
- Heapsort is an efficient sorting procedure that requires $O(N \log N)$ time.
- Conceptually heapsort consists of making $N$ insertions into a heap, followed by $N$ removals.
- Heapsort can be made to run faster by applying the trickle-down algorithm directly to $N/2$ items in the unsorted array, rather than inserting $N$ items.
- The same array can be used for the initial unordered data, for the heap array, and for the final sorted data. Thus heapsort requires no extra memory.