Binary Search Trees

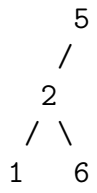
Today we consider a specific type of binary tree in which there happens to be an ordering defined on the set of elements the nodes. If the elements are numbers then there is obviously an ordering. If the elements are strings, then there is also a natural ordering, namely the dictionary ordering, also known as "lexicographic ordering".

Definition: binary search tree

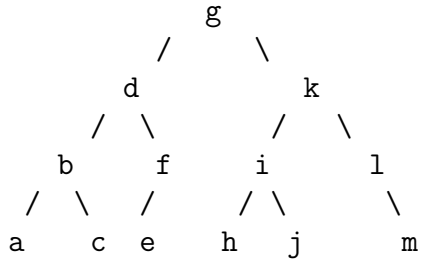
A *binary search tree* is a binary tree such that

- each node contains an element, called a *key*, such that the keys are comparable, namely there is a strict ordering relation < between keys of different nodes
- any two nodes have different keys (i.e. no repeats, i.e. duplicates)
- for any node,
  - all keys in the left subtree are less than the node's key
  - all keys in the right subtree are greater than the node's key.

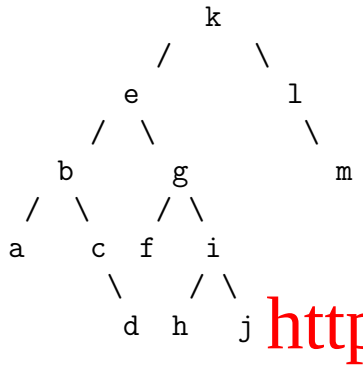
  Note this is stronger than just saying that the left child's key is less than the node's key which is less than the right child's key. For example, the following is not a binary search tree.



One important property of binary search trees is that *an inorder traversal of a binary search tree gives the elements in their correct order.* Here is an example with nodes containing keys `abcdefghijklm`. Verify that an inorder traversal gives the elements in their correct order.



Here is another example with the same set of keys:



BST as an abstract data type (ADT)

One performs several common operations on binary search trees:

- `find(key)`: given a key, return a reference to the node containing that key (or null if key is not in tree)

- `findMin()` or `findMax()`: find the node containing the smallest or largest key in the tree and return a reference to the node containing that key

- `add(key)`: insert a new node into the tree such that the node contains the key and the node is in its correct position (if the key is already in the tree, then do nothing)

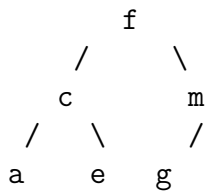
- `remove(key)`: remove from the tree the node containing the key (if it is present)

Here are algorithms for implementing these operations.

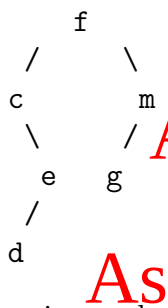
```
find(root,key){                    //    returns a node
  if (root == null)
     return null
  else if (root.key == key))
    return root
  else if (key < root.key)
     return  find(root.left, key)
  else
     return  find(root.right, key)
}

findMin(root){              //  returns a node
   if (root == null)        // only necessary for the first call
     return null
   else if (root.left == null)
      return root
   else
      return findMin(root.left)
}
```

For example, here the minimum key is a.



Notice however that the minimum key is not necessarily a leaf i.e. it can occur if the key has a right child but no left child. Here the minimum key is c:



The reasoning and method is similar for findMax.

```
findMax(root){                    // returns a node
   if (root == null)
      return null
   else if (root.right == null))
      return root
   else
      return findMax(root.right)
}
```


Let's next consider adding (inserting) a key to a binary search tree. If the key is already there, then do nothing. Otherwise, make a new node containing that key, and insert that node into its *unique* correct position in the tree.

```
add(root,key){      // returns root
   if (root == null)              //  base case:
      root =  new BSTnode(key)    //  makes a new node and returns it
   else if (key < root.key){
      root.left = add(root.left,key)
   else if (key > root.key){
      root.right = add(root.right,key)
   return root
}
```

The new node is always added at a leaf and so the base case is always reached eventually. For the non-base case, the situation is subtle. The code says that a new node is added to either the left or right subtree and then the reference to the left or right subtree is re-assigned. It is reassigned

to the root of the left or right subtree, which has the new node added it. Why is it necessary to reassign the reference like that?

Suppose we add the new node to the left subtree. If the new node is a descendent of the root node of the left subtree , then the assignment `root.left = add(root.left,key)` doesn't change the reference `root.left` in the `root` node; it just assigns it to the same node it was referencing beforehand. *However,* if `root.left` was null and then the new node was added to the left subtree of `root`, then the call `add(root.left,key)` will create and return the new node, namely it is the root of a subtree with one node. If we don't assign that node to `root.left` as the code says, then the new node that is created will not in fact be added to the tree.

Next, consider the problem of removing a node from a binary search tree.


```
remove(root, key){                    // returns root
    if( root  == null )
         return null
    else if ( key < root.key )                                    (*)
         root.left = remove( root.left, key )
    else  if ( key > root.key )                                   (*)
         root.right = remove( root.right, key)
    else  if root.left == null                                    (**)
         root  =  root.right          // or just "return root.right"
    else  if root.right == null                                   (**)
         root  = root.left            // or just "return root.left"
    else{                                                         (***)
         root.key = findMin( root.right).key
         root.right = remove( root.right, root.key )
    }
    return root
}
```


The (*) conditions handle the case that the key is not at the root, and in this case, we just recursively remove the key from the left or right subtree. Note that we replace the left or right subtree with a subtree that doesn't contain the key. To do this, we use recursion, namely we remove the key from the left or right subtree.

The more challenging case is that the key that we want to remove is at the root (perhaps after a sequence of recursive calls). In this case we need to consider four possibilities:

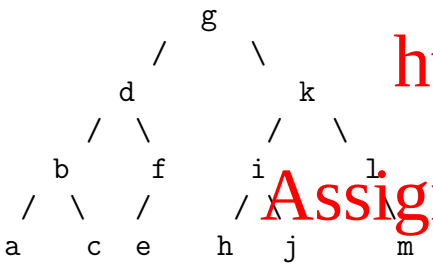
- the root has no left child
- the root has no right child
- the root has no children at all
- the root has both a left child and a right child

In the first two cases – see (**) in the algorithm – we just replace the root node with the subtree in the non-empty child. Note that the third case is accounted for here as well; since both children are null, the root will become null.

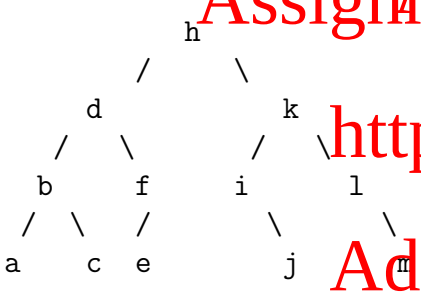
In the fourth case – (***) – we take the following approach. We replace the root with the minimum node in the right subtree. It does so in two steps. It copies the key from the smallest node in the right subtree into the root node, and then it removes the smallest node node from the the right subtree.

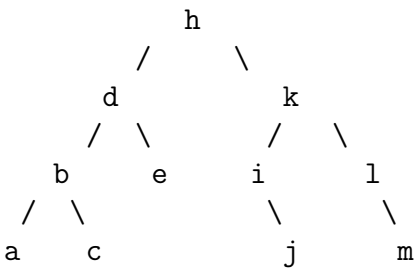
Example (remove)

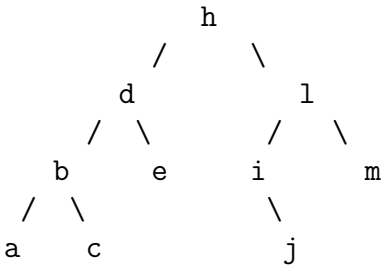
Take the following example with nodes `abcdefghijklm`:



and then remove elements `g`, `f`, `k` in that order.







Let's consider the best and worst case performance. The best case performance for this data structure tends to occur when the keys are arranged such that the tree is balanced, so that all leaves are at the same depth (or depths of two leaves differ by at most one). However, if one adds keys into the binary tree and doesn't rearrange the tree to keep it balanced, then there is no guarentee

that the tree will be anywhere close to balanced, and in the worst case a BST with n nodes could have height $n-1$. This implies the following best and worst cases:

Operations/Algorithms for Lists	Best case, $t_{best}(n)$	Worst case, $t_{worst}(n)$
findMax(), findMin()	$\Theta(1)$	$\Theta(n)$
find(key)	$\Theta(1)$	$\Theta(n)$
add(key)	$\Theta(1)$	$\Theta(n)$
remove(key)	$\Theta(1)$	$\Theta(n)$

In COMP 251, you will learn about balanced binary search trees, e.g. AVL trees or red-black trees. If a tree is balanced, then the operations have all $\Theta(\log n)$. This is an interesting topic, but technically involved so I won't say anything more about it here.

Priority Queue

Recall the definition of a queue. It is a collection where we remove the element that has been in the collection for the longest time. Alternatively stated, we remove the element that first entered the collection. A natural way to implement such a queue was using a linear data structure, such as a linked list or a (circular) array.

A *priority queue* is a different kind of queue, in which the next element to be removed is defined by a priority, which is a more general criterion. For example, in a hospital emergency room, patients are treated not in a first-come first-serve basis, but rather by the urgency of the case. To define the next element to be removed, it is necessary to have some way of comparing any two objects and deciding which has greater priority. Once a comparison method is chosen for determining priority, *the next element to be removed is the one with greatest priority.* Heads up: with priority queues, one typically assigns low numerical values to high priorities. Think "my number one priority", "my number 2 priority", etc.

One way to implement a priority queue of elements (often called *keys*) is to maintain a sorted list. This could be done with a linked list or array list. Each time a element is added, it would need to be inserted into the sorted list. If the number of elements were huge, however, then this would be an inefficient representation since in the worst case the add and remove would be $O(n)$.

Heaps

The usual way to implement a priority is to use a data structure called a *heap*. To define a heap, we first need to define a complete binary tree. We say a binary tree of height h is *complete* if every level l less than h has the maximum number (2^l) of nodes, and in level h all nodes are as far to the left as possible. A *heap* is a complete binary tree, whose nodes are comparable and satisfy the property that *each node is less than its children*. (To be precise, when I say that the nodes are comparable, I mean that the *elements* are comparable, in the sense that they can all be ordered – recall the *Comparable* interface in Java, presented in lecture 16) This is the default definition of a heap, and is sometimes called a *min heap*.

A *max heap* is defined similarly, except that the element stored at each node is greater than the elements stored at the children of that node. Unless otherwise specified, we will assume a min heap in the next few lectures. Note that it follows from the definition that the smallest element in a heap is stored at the root.

As with stacks and queues, the two main operations we perform on heaps are `add` and `remove`.

add

To add an element to a heap, we create a new node and insert it in the next available position of the complete tree. If level h is not full, then we insert it next to the rightmost leaf. If level h is full, then we start a new level at height $h + 1$.

Once we have inserted the new node, we need to be sure that the heap property is satisfied. The problem could be that the parent of the node is greater than the node. This problem is easy to solve. We can just swap the element of the node and its parent. We then need to repeat the

same test on the new parent node, etc, until we reach either the root, or until the parent node is less than the current node. This process of moving a node up the heap, is often called "upheap".

```
add(element){
   cur = new node at next leaf position
   cur.element = element
   while (cur != root) && (cur.element < cur.parent.element){
       swapElement(cur, parent)
       cur = cur.parent
   }
}
```


You might ask whether swapping the element at a node with its parent's element can cause a problem with the node's sibling (if it exists). No, it cannot. Before the swap, the parent is less than the sibling.[1] So if the current node is less than its parent, then the current node must be less than the sibling too. So, swapping the node's element with its parent's element preserves the heap property with respect to the node's current sibling.

For example, suppose we have a heap with two elements, e and g. Then we add an element to the * position below and we find that * < e. So we swap them. But if * < e then * < g.


```
  e
/   \
g    *
```


Here is a bigger example. Suppose we add element c to the following heap.

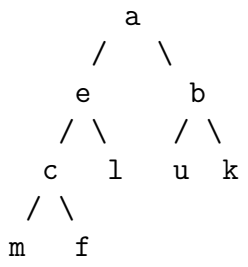

```
          a
        /   \
       e     b
      / \   / \
     f   l u   k
    /
   m
```

We add a node which is a sibling to m and assign c as the element of the new node.

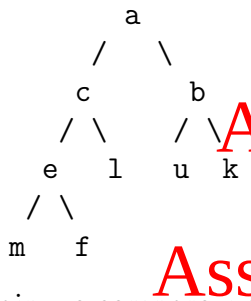
```
          a
        /   \
       e     b
      / \   / \
     f   l u   k
    / \
   m   c
```

Then we observe that c is less than f, the element of its parent, so we swap c,f to get:

[1] Here I say that one node is less than another, but what I really mean is that the element at one node is less than the element at the other node.



Now we continue up the tree. We compare c with its new parent's element e, see that the elements need to be swapped, and swap them to get:



Again we compare c to its parent. Since c is greater than a, we stop and we're done.

removeMin

Next, let's look at how we remove elements from a heap. Since the heap is used to represent a priority queue, we remove the minimum element, which is the root.

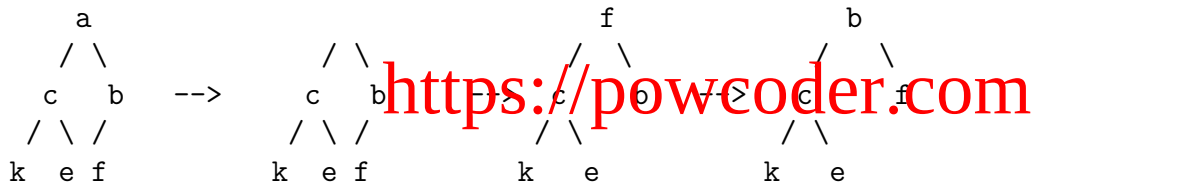
How do we fill the hole that is left by the element we removed ? We first copy the last element in the heap (the rightmost element in level h) into the root, and delete the node containing this last element. We then need to manipulate the elements in the tree to preserve the heap property that each parent is less than its children.

We start at the root, which contains an element that was previously the rightmost leaf in level h. We compare the root to its two children. If the root is greater than at least one of the children, we swap the root with the smaller child. Moving the smaller child to the root does not create a problem with the larger child, since the smaller child is smaller than the larger child.

```
removeMin(){                  // returns smallest element
   tmp = root.element
   remove last leaf node and put its element into the root
   cur = root
   while ((cur has a left child) and
          ((cur.element > cur.left.element) or
           (cur has a right child and cur.element > cur.right.element)))
       minChild = child with the smaller element
       swapElement(cur, minChild)
       cur = minChild
   }
   return tmp
}
```

The condition in the while loop is rather complicated, and you may have just skipped it. Don't. There are several possible events that can happen and you need to consider each of them. One is that the current node has no children. In that case, there is nothing to do. The second is that the current node has one child, in which case it is the left child. The issue here is that the left child might be smaller. The third is that the current node might have two children. In that case, one of these two children *has to be* smaller than the current node. (See Exercises.)

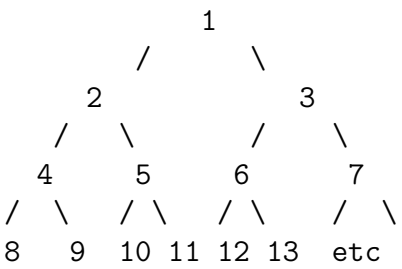
Here is an example:



If we apply `removeMin()` again and again until all the elements are gone, we get the following sequence of heaps with elements removed in the following order: `b, c, e, f, k`.

Implementing a heap using an array

A heap is defined to be a complete binary tree. If we number the nodes of a heap by a level order traversal and start with index 1, rather than 0, then we get an indexing scheme as shown below.



These numbers are NOT the elements stored at the node. Rather we are just numbering the nodes so we can index them.

The idea is that an array representation defines a simple relationship between a tree node's index and its children's index. If the node index is i, then its children have indices $2i$ and $2i+1$. Similarly, if a non-root node has index i then its parent has index $i/2$.

add(element)

Suppose we have a heap with `k` elements which is represented using an array, and now we want to add a `k+1`-th element. Last lecture we sketched an algorithm doing so. Here I'll re-write that algorithm using the simple array indexing scheme. Let `size` be the number of elements in the heap.

These elements are stored in array slots 1 to `size`, i.e. recall that slot 0 is unused so that we can use the simple relationship between a child and parent index.

```
add(element ){
  size = size + 1          // number of elements in heap
  heap[ size ] = element   // assuming array has room for another element
  i = size

//  the following is sometimes called "upHeap"  -- see below

  while (i > 1  and heap[i] < heap[ i/2 ])){
    swapElements( i, i/2 )
    i = i/2
  }
}
```


Example

Suppose we have a heap with eight characters and we add one more, a `c`.

```
1   2   3   4   5   6   7   8   9
--------------------------------------
a   e   b   f   l   u   k   m   c
a   e   b   c   l   u   k   m   f  <----  c swapped with f (slots 9 & 4)
a   c   b   e   l   u   k   m   f  <----  c swapped with e (slots 4 & 2)
```

At the end of last lecture we showed how a heap could be represented using an array, and we rewrote the `add` method using the array representation instead of the binary tree representation. Today we will examine how to build a heap by repeatedly calling the `add` method on a list of elements, and we will analyze the time complexity of building a heap. We will then review the `removeMin` method and rewrite it in terms of the array representation. Finally we will put this all together and show how to sort a list of elements using a heap – called *heapsort*.

Building a heap

We can use the `add` method to build a heap as follows. Suppose we have a list of `size` elements and we want to build a heap.

```
buildHeap(list){
   create an empty heap
   for (k = 0; k < list.size; k++)
        heap.add( list[k] )
   return heap
}
```

We can write this in a slightly different way.

```
buildHeap(list){
  create an array with list.size+1 slots
   for (k = 1; k <= list.size; k++){
       heap[k] = list[k-1]       //  say list indices are 0, .. ,size-1
       upHeap(heap, k)
   }
}
```

where `upHeap(heap, k)` is defined as follows.

```
upHeap(heap, k){
    i = k
    while (i > 1  and heap[i] < heap[ i/2 ]){
       swapElements( i, i/2 )
       i = i/2
    }
}
```


Are we sure that the buildHeap method indeed builds a heap? Yes, and the argument is basic mathematical induction. Adding the first element gives a heap with one element. If adding the first k elements results in a heap, then adding the $k+1$-th element also results in a heap since the upHeap method ensures this is so.

Time complexity

How long does it take to build a heap in the best and worst case? Before answering this, let's recall some notation. We have seen the "floor" operation a few times. Recall that it rounds down to the nearest integer. If the argument is already an integer then it does nothing. We also can define the ceiling, which rounds up. It is common to use the following notation:

- $\lfloor x \rfloor$ is the largest integer that is less than or equal to x. $\lfloor\ \rfloor$ is called the *floor* operator.
- $\lceil x \rceil$ is the smallest integer that is greater than or equal to x. $\lceil\ \rceil$ is called the *ceiling* operator.

Let i be the index in the array representation of elements/nodes in a heap, then i is found at level *level* in the corresponding binary tree representation. The level of the corresponding node i in the tree is such that

$$2^{level} \leq i < 2^{level+1}$$

or

$$level \leq \log_2 i < level + 1,$$

and so

$$level = \lfloor \log_2 i \rfloor.$$

We can use this to examine the best and worst cases for building a heap.

In the best case, the node i that we add to the heap satisfies the heap property immediately, and no swapping with parents is necessary. In this case, building a heap takes time proportional to the number of nodes n. So, best case is $O(n)$.

What about the worst case? Since $level = \lfloor \log_2 i \rfloor$, when we add element i to the heap, in the *worst case* we need to do $\lfloor \log_2 i \rfloor$ swaps up the tree to bring element i to a position where it is less than its parent, namely we may need to swap it all the way up to the root. If we are adding n nodes in total, the worst case number of swaps is:

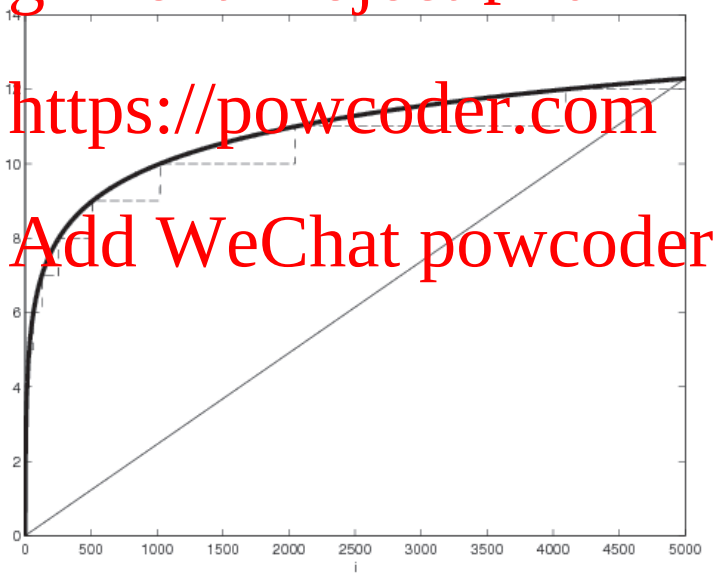
$$t(n) = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

To visualize this sum, consider the plot below which show the functions $\log_2 i$ (thick) and $\lfloor \log_2 i \rfloor$ (dashed) curves up to $i = 5000$. In this figure, $n = 5000$.

The area under the dashed curve is the above summation. It should be visually obvious from the figures that

$$\frac{1}{2} n \log_2 n < t(n) < n \log_2 n$$

where the left side of the inequality is the area under the diagonal line from (0,0) to $(n, \log_2 n)$ and the right side $(n \log_2 n)$ is the area under the rectangle of height $\log_2 n$. From the above inequalities, we conclude that in the worst of building a heap is $O(n \log_2 n)$.



removeMin

Next, recall the `removeMin()` algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```
removeMin(){
    element = heap[1]            //   heap[0] not used.
    heap[1] = heap[size]
    heap[size] = null
```

```
    size = size - 1
    downHeap(1, size)       //   see next page
    return element
}
```

This algorithm saves the root element to be returned later, and then moves the element at position `size` to the root. The situation now is that the two children of the root (node 2 and node 3) and their respective descendents each define a heap. But the tree itself typically won't satisfy the heap property: the new root will be greater than one of its children. In this typical case, the root needs to move down in the heap.

The `downHeap` helper method moves an element from a starting position in the array down to some maximum position in the heap. I will use this helper method in a few ways in this lecture.

```
downHeap(start,maxIndex){      // move element from starting position
                               //  down to at most position maxIndex
   i = start
   while (2*i <= maxIndex){ leftchild = 2*i // if there is a left child
      child = 2*i
      if (child < maxIndex) {           // if there is a right sibling
         if (heap[child + 1] < heap[child])
                                        // if rightchild < leftchild ?
         child = child + 1
      }
      if (heap[child] < heap[i]){ // swap with child?
         swapElements(i , child)
         i = child
      }
      else  break    // exit while loop
   }
}
```


This is essentially the same algorithm we saw last lecture. What is new here is that (1) I am expressing it in terms of the array indices, and (2) there are parameters that allow the downHeap to start and stop at particular indices.

Heapsort

A heap can be used to sort a set of elements. The idea is simple. Just repeatedly remove the minimum element by calling `removeMin()`. This naturally gives the elements in their proper order.

Here I give an algorithm for sorting "in place". We repeatedly remove the minimum element the heap, reducing the size of the heap by one each time. This frees up a slot in the array, and so we insert the removed element into that freed up slot.

The pseudocode below does exactly what I just described, although it doesn't say "`removeMin()`. Instead, it says to swap the root element i.e. `heap[1]` with the last element in the heap i.e. `heap[size+1-i]`. After `i` times through the loop, the remaining heap has `size - i` elements,

and the last `i` elements in the array hold the smallest `i` elements in the original list. So, we only downheap to index `size - i` .

```
heapsort(list){
   buildheap(list)
   for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
   }
   return reverse(heap)
}
```


The end result is an array that is sorted from largest to smallest. Since we want to order from smallest to largest, we must reverse the order of the elements.

What is the time complexity of heapsort? If we have n elements, then we have to go through the for loop n times. Each time through, we need to go down at most the height of the tree (and for many elements, we don't have to go that far). Since the height of the tree is about $\log_2 n$, in the worst case, we need to go $O(n \log n)$. The final step, where we reverse the elements in the array only needs to be done once. It is done in $O(n)$ time, by swapping i and $n+1-i$ for $i = 1$ to $\frac{n}{2}$. So heapsort in the worst case is $O(n \log_2 n)$.

Example

The example below shows the state of the array after each pass through the for loop. The vertical line marks the boundary between the remaining heap (on the left) and the sorted elements (on the right).


```
1   2   3   4   5   6   7   8   9
-------------------------------------
a   d   b   e   l   u   k   f   w |
b   d   k   e   l   u   w   f | a   (removed a, put w at root, ...)
d   e   k   f   l   u   w | b   a   (removed b, put f at root, ...)
e   f   k   w   l   u | d   b   a   (removed d, put w at root, ...)
f   l   k   w   u | e   d   b   a   (removed e, put u at root, ...)
k   l   u   w | f   e   d   b   a   (removed f, put u at root, ...)
l   w   u | k   f   e   d   b   a   (removed k, put w at root, ...)
u   w | l   k   f   e   d   b   a   (removed l, put u at root, ...)
w | u   l   k   f   e   d   b   a   (removed u, put w at root, ...)
w   u   l   k   f   e   d   b   a   (removed w, and done)
```

Note that the last pass through the loop doesn't do anything since the heap has only one element left (`w` in this example), which is the largest element. We could have made the loop go from `i = 1 to size - 1`.

What if elements are already sorted (or nearly so)?

You might think the best case for heapsort is that the elements in the list are already in order, since in that case the buildheap step is $O(n)$ rather than $O(n \log_2 n)$, i.e. there are no swaps in the build heap step since each element is compared to its parent and found to be greater than its parent.[2] However, the $n*$downHeap part of the algorithm would be as slow as possible since when the elements are put into the root they greater than all the remaining elements and will need to be downheaped all the way to a leaf.

What if the elements are in the opposite order to begin with? Will heapsort perform better than $O(n \log_2 n)$ in this case? No, it won't. The buildheap step will take time $O(n \log_2 n)$ since each element that is added needs to be swapped all the way to the root.

Discussion...

Why is quicksort in practice preferred over heapsort? First, one needs to note that real quicksort implementations do not choose the last element of the list as the pivot. The reason is that choosing the last element would lead to $O(n^2)$ performance if the list is already sorted or nearly so, and this case does come up in practice sometimes. For example, a better pivot choice is to take the first, middle, and last elements of the list, and choose the median value of these three. If the list is sorted forward or sorted backwards to begin with, then choosing the "median of three" as it is called will split in the middle (yipeee!). And if the list has random order, then the median of three will be more likely to be close to the actual median than if one just chose one of the three. Of course, determining which of these three chosen ones *is* the median requires a bit of extra work for every split, and so it increases the constant factor at each step of the recursion.

This is related to a question that was asked in class: why doesn't heapsort or quicksort just check at the start if the list is already sorted, which would take time only $O(n)$. Yes, you could do that. And it might make sense to do that if this case did arise in practice often enough. But if this case arises only rarely, then this would be extra work that you need to do every time, but that would benefit you rarely.

[2] In the lecture itself, I rushed through this case and mistakenly said that the buildheap part would be $O(n \log_2 n)$ in this case.