CS 61B Data Structures and Programming Methodology

July 22, 2008

David Sun

Hash Tables and Binary Search Trees

- A dictionary is used to look up *arbitrary* <key, value> pairs, *given* a *specific* key to search
  - With a good hash code and compression function we can do look up in constant time.
  - But there is no notion of priority or ordering among the keys so to find the smallest/largest element you will need to compare all the elements.

Binary Search Tree

- A binary search tree stores <key, value> pairs with a notion of order:
  - Looking up an arbitrary element may be slower, but we can perform query based searches.
  - What's the smallest element in the tree?
  - What's the largest element in the tree?
  - These operations are *O(logN)* where N is the number of elements.

What if we are only interested the smallest or largest element ?

Priority Queues

- A priority queue is used to prioritize entries.
  - Just like binary search tree a total order is defined on the keys.
  - But you can identify or remove the entry whose key is the largest (or the smallest) in *O(1)* time.
- Application:
  - Schedule jobs on a shared computer. The priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest priority job is selected from those pending.

Main Operations

- `insert`: adds and entry to the priority queue.
- `max`: returns the entry associated with the maximum key (without removing it).
- `removeMax`: removes and returns the entry associated with the maximum key.

```
public interface PriorityQueue {
    public int size();
    public boolean isEmpty();
    Entry insert(Object k, Object v);
    Entry max();
    Entry removeMax();
}
```

Binary Heaps

- A Binary Heap is a binary tree, with two additional properties
  - Shape Property: It is a complete binary tree – a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right.
  - Heap Property (or Heap Order Property): *No child has a key greater than its parent's key.* This property is applied recursively: any subtree of a binary heap is also a binary heap.
- If we use the notion of *smaller than* in the Heap Property we get a *min-heap.* We'll look at *max-heap* in this class.

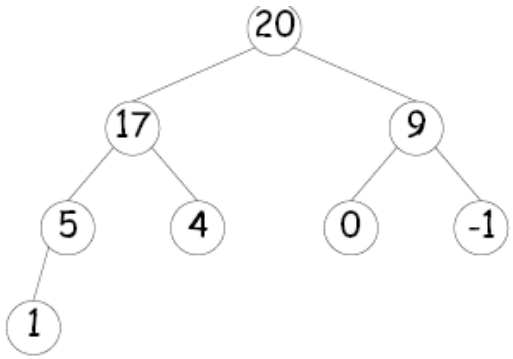
`max()`

- Trivial: The heap-order property ensures that the entry with the maximum key is always at the *top* of the heap. Hence, we simply return the entry at the root node.
  - If the heap is empty, return null or throw an exception.
- Runs in Theta(1) time.

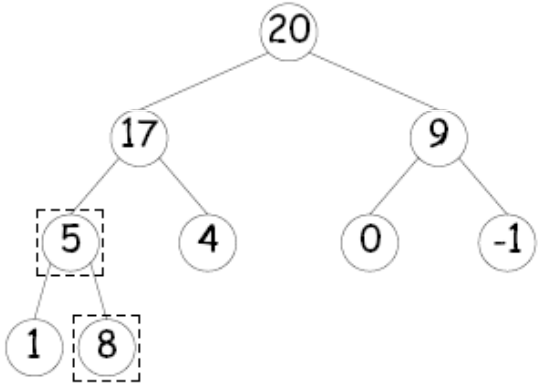
insert()

Let *x* be the new entry *(k, v)*.

1. Place the new entry *x* in the bottom level of the tree, at the first free spot from the left. If the bottom level is full, start a new level with *x* at the far left.
2. If the new entry's key violates the heap-order property then compare *x*'s key with its parent's key; if *x*'s key is larger, we exchange *x* with its paren. Repeat the procedure with *x*'s new parent.

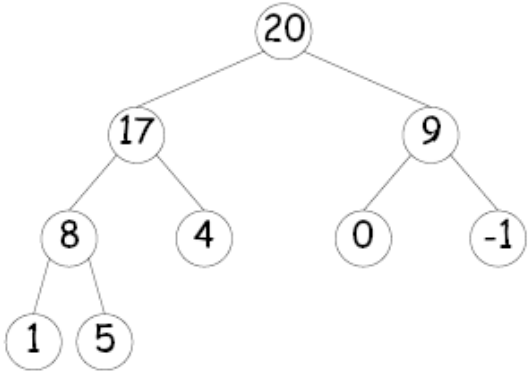


Original



Inserting <8,v>

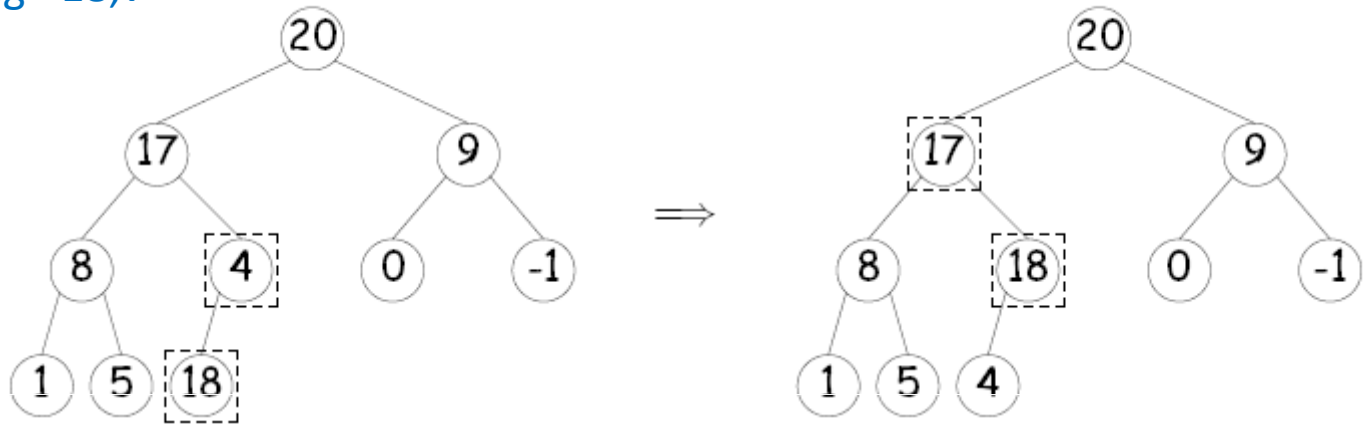
Dashed boxes show where the heap property violated



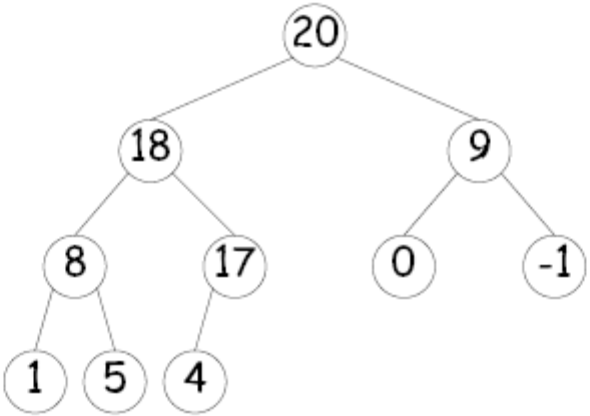
Re-heapify up

insert()

Inserting <18,v>



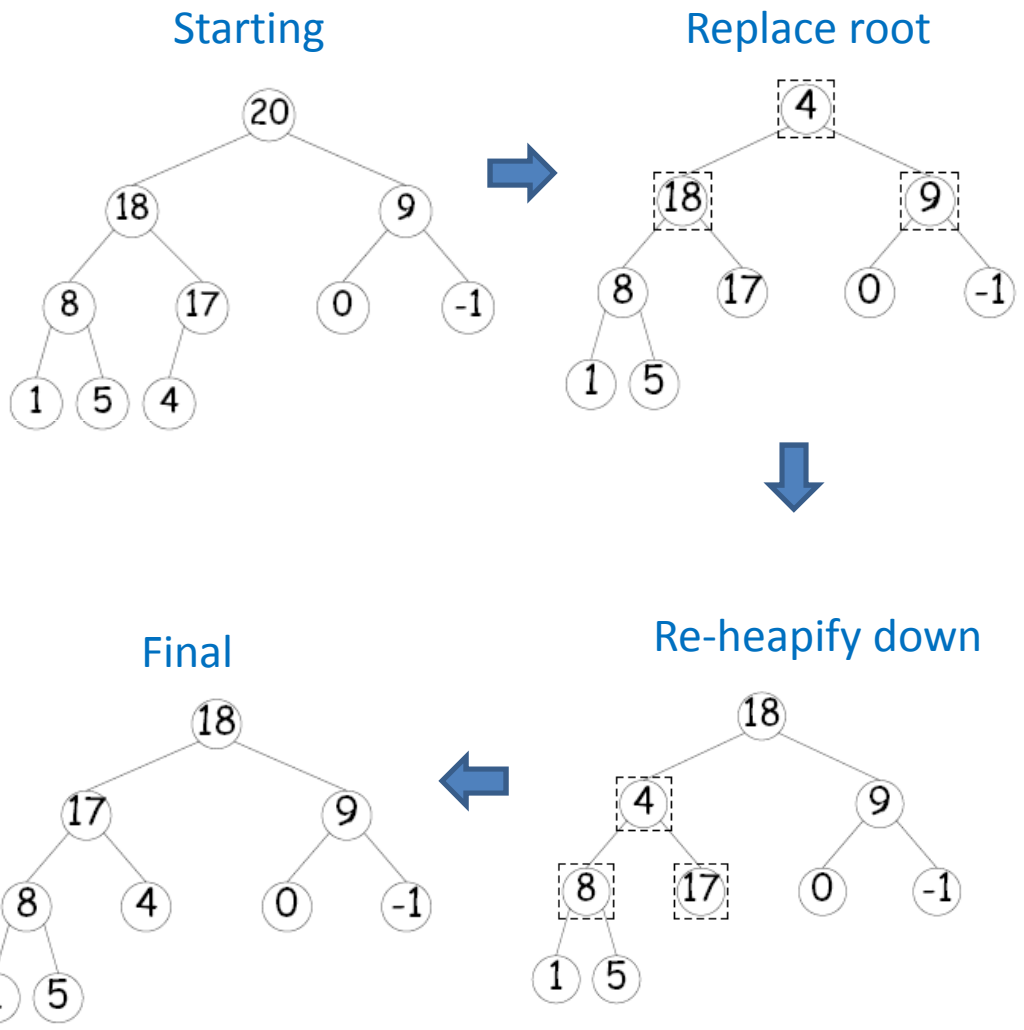
⇓



What's the time complexity of `insert`*?*

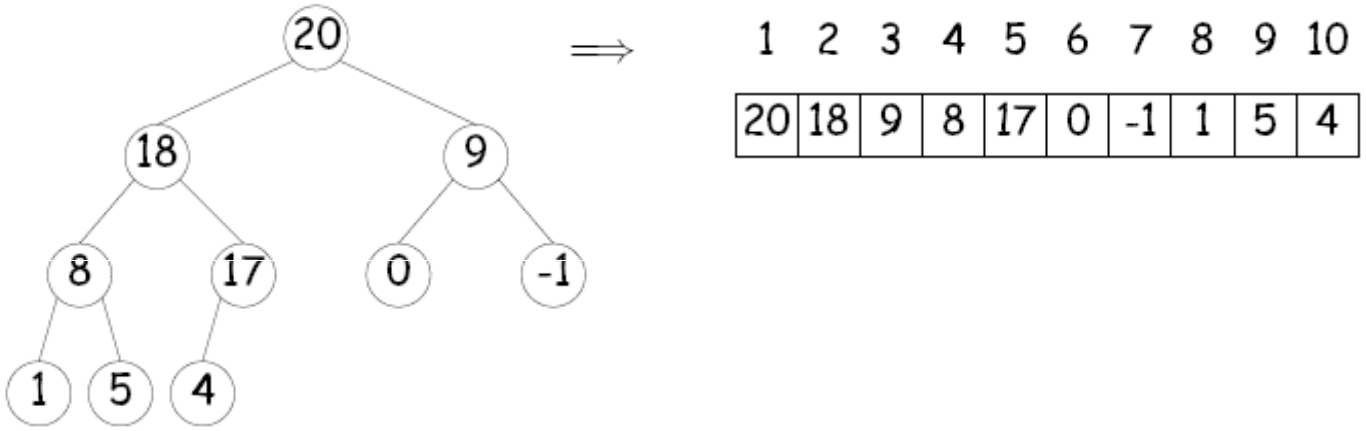
removeMax()

1. If the heap is empty, return null or throw an exception.
2. Otherwise, remove the entry at the root node. Replace the root with the last entry in the tree x, so that the tree is still complete.
3. If the root violates the heap property then compare *x* with it's children, swap *x* with the child with the **larger** key, repeat until *x* is greater than or equal to its children or reach a leaf.



Storing Binary Heap

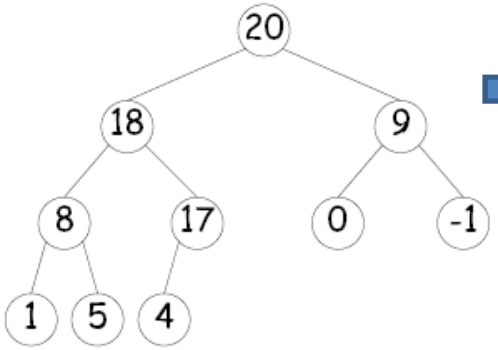
- Since heaps are complete, one can use arrays for a compact representation



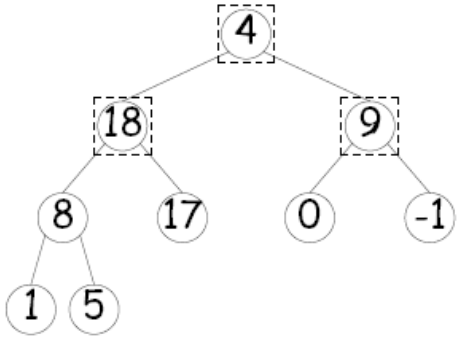
- No node needs to store explicit references to its parent or children
  - If a node's index is *i*, its children's indices are *2i* and *2i+1*, and its parent's index is *floor(i/2).*

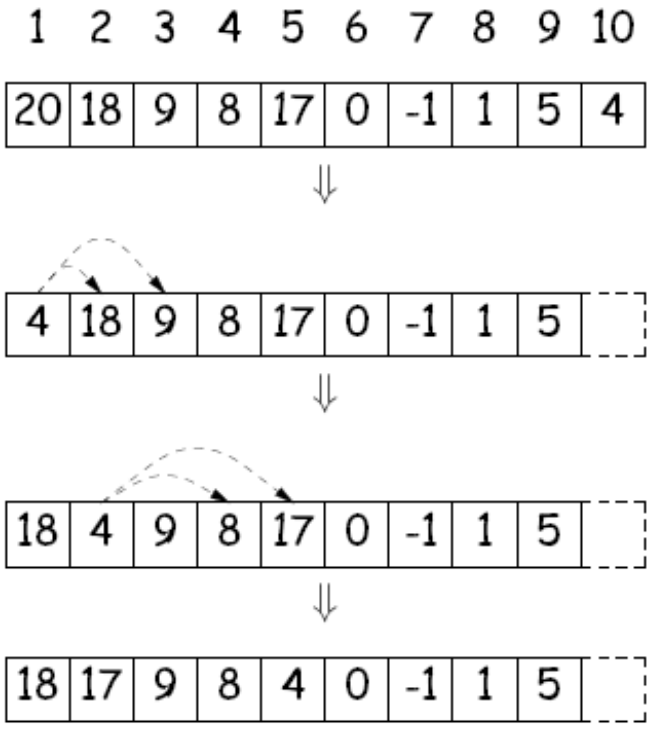
removeMax()

Starting

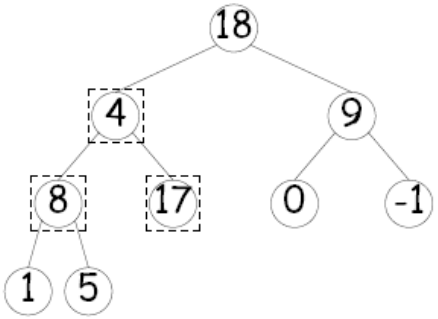


Replace root

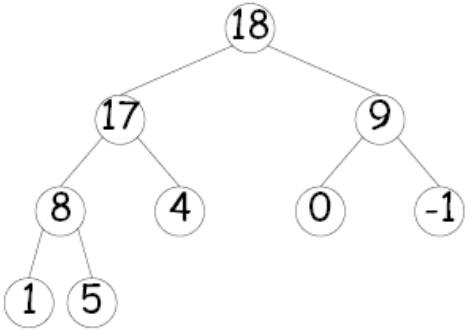




Re-heapify down



Final



Running Times

- We could use a list or array, sorted or unsorted, to implement a priority queue. The following table shows running times for different implementations, with *n* entries in the queue.

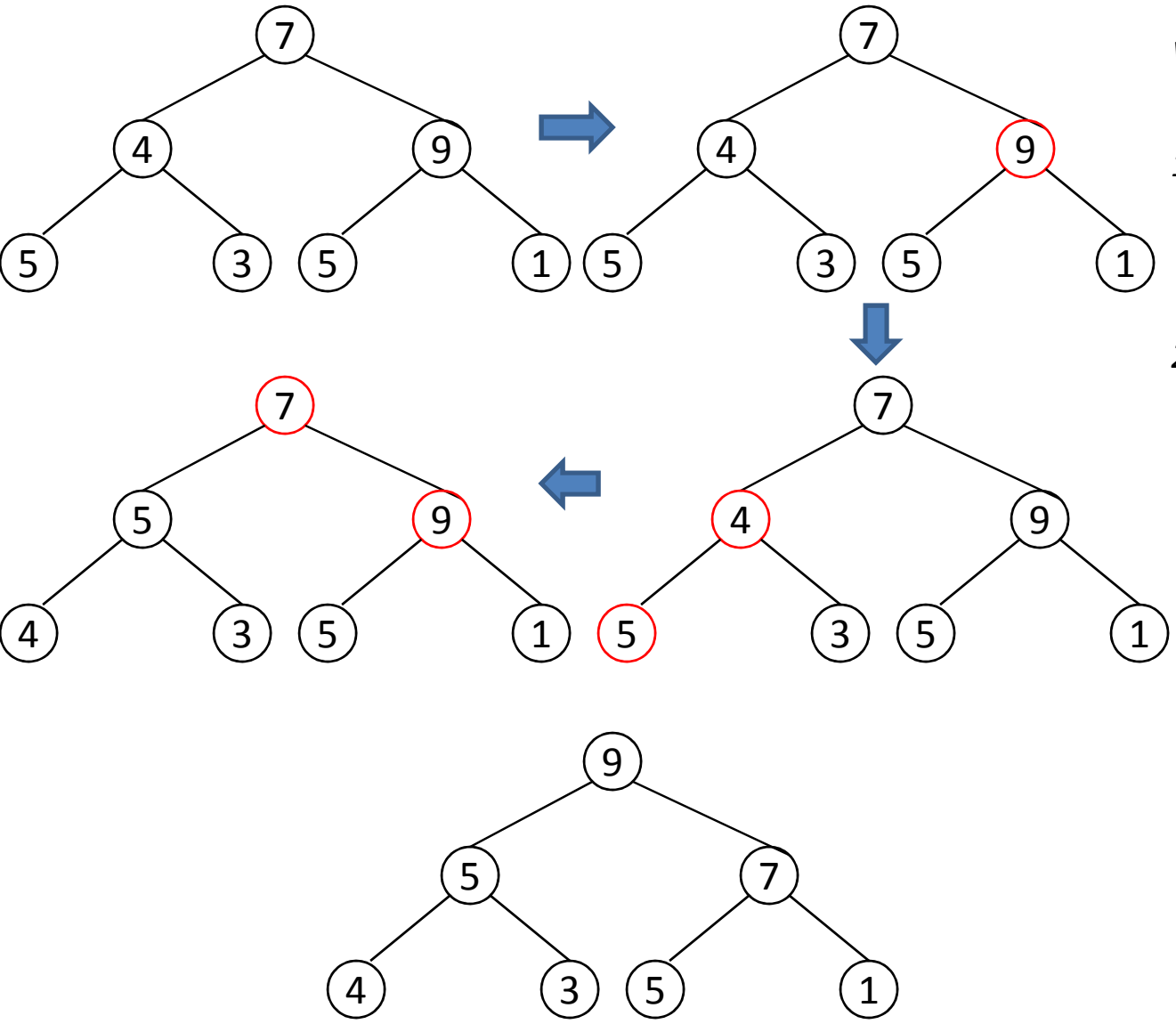
	Binary Heap	Sorted List/Array	Unsorted List/Array
max	Theta(1)	Theta(1)	Theta(n)
insert (worst-cast)	Theta(log n)*	Theta(n)	Theta(1)*
insert(best-case)	Theta(1)*	it depends	Theta(1)*
removeMax (worst)	Theta(log n)	Theta(1)	Theta(n)
removeMax (best)	Theta(1)	Theta(1)	Theta(n)

* If you are using an array-based data structure, these running times assume that you don't run of room. If you do, it will take Omega(n) time to allocate a larger array and copy them into it.

Bottom-Up Heap Construction

- Suppose we are given a bunch of randomly ordered entries, and want to make a heap out of them.
- What's the obvious way
  - Apply `insert` to each item in *O(n log n)* time.
- A better way: `bottomUpHeap()`
  1. Make a complete tree out of the entries, in any *random* order.
  2. Start from the last internal node (non-leaf node), in reverse order of the level order traversal, *heapify down* the heap as in `removeMax`().

Example



Why this works? We can argue inductively:

1. *The leaf nodes satisfy the heap order property vacuously (they have no children).*
2. *Before we bubble an entry down, we know that its two child subtrees **must be** heaps. Hence, by bubbling the entry down, we create a larger heap rooted at the node where that entry started.*

Cost of Bottom Up Construction

- If *each* internal node bubbles *all the way* down, then the running time is proportional to the sum of the heights of all the nodes in the tree.
- Turns out this sum is less than n, where n is the number of entries being coalesced into a heap.
- Hence, the running time is in *O(n)*, which is better than inserting n entries into a heap individually.

Other Types of Heaps

- Binary Heap is not the only implementation of a priority queue, other heaps also exist.
- Several important variants are called "mergeable heaps", because it is relatively fast to combine two mergeable heaps together into a single mergeable heap.
- The best-known mergeable heaps are called "binomial heaps," "Fibonacci heaps," "skew heaps," and "pairing heaps.“
- We will not examine these in CS61B, but it’s good to know that they exist.

Sorting

Sort

- Sorting supports basic searching
  - For a number in a phone book
  - A website with the most relevant information to your search query.
- Also supports other kinds of search:
  - Are there two equal items in this set?
  - Are there two items in this set that both have the same value for property X?
  - What are my nearest neighbors?
- Sorting is perhaps the simplest fundamental problem that offers a large variety of algorithms, each with its own inherent advantages and disadvantages.

Insertion Sort

- Simple idea:
  - Starting with empty sequence of outputs *S* and the unsorted list of n input items *I.*
  - Add each item from input *I*, inserting into output sequence *S* at a position so the output is still in sorted order
  - Invariant: at the *k*th iteration, the elements from *0* to *k-1* in the output sequence *S* are sorted.
- Example

Insertion Sort

- If destination is a linked list
  - Theta(n) worst-case time to find the right position of *S* to insert each item.
  - Theta(1) time to insert the item.
- If destination is an array
  - Find the right position in *S* in O(log n) time by binary search.
  - Theta(n) worst-case time to shift the larger items over to make room for the new item.
- In either case, insertion sort is only an $O(N^2)$ algorithm- but for a different reason in each case.
- Good for small sets of data.

In-place Sort with Array

- If *S* is an array, we can do an in place sort:
  - store sorted items in the same array that initially held the input items, and uses only O(1) or perhaps O(log n) additional memory (in addition to the input array).
- To do an in-place insertion sort
  - partition the array into two pieces: the left portion (initially empty) holds *S*, and the right portion holds I.
  - With each iteration, the dividing line between *S* and *I* moves one step to the right.
- Example

Selection Sort

- Simple idea:
  - Starting with empty iteration, the first *k* elements of the input are sorted. outputs and the unsorted list of n input items
  - Walk through the input and find the smallest item and append to the end of the output.
  - Invariant: at the *k*th
- Whether *S* is an array or linked list, finding the smallest item takes Theta(n) time, so selection sort takes Theta(n^2) time, even in the best case!
- Hence, it's even worse than insertion sort.
- If *S* is an array, we can do an in-place selection sort. After finding the item in S having smallest key, swap it with the first item in S
- *Example.*

Reading

- Objects, Abstraction, Data Structures and Design using Java 5.0
  - Chapter 8: pp434 - 440