Priority Queues
Heap Algorithms

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, April 23, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview
Tables & Priority Queues

Major Topics

- ✓ Introduction to Tables
- Priority Queues
- Heap algorithms
- Heaps and Priority Queues in practice
- 2-3 Trees
- Other balanced search trees
- Hash Tables
- Prefix Trees
- Tables in practice

Review
Introduction to Tables [1/4]

Our ultimate value-oriented ADT is **Table**.

What do we use a Table for?

- To hold **sparse** array data.
  - `arr[6] = 1; arr[1000000000] = 2;`
- To hold "arrays" whose indices are not nonnegative integers.
  - `arr2["hello"] = 3;`
- To hold data accessed by key fields. For example:
  - Customers accessed by phone number.
  - Students accessed by student ID number.
  - Any other kind of data with an ID code.

How do we implement a Table? We know lots of lousy ways:

- Using a Sequence of key-data pairs.
  - Array-based or Linked-List-based.
  - Sorted or unsorted.
- Using a Binary Search Tree holding key-data pairs.

Review
Introduction to Tables [2/4]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced*** Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Amort. const. (or constant)*	Linear**	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear**	Linear**	Linear	Logarithmic

*Amortized constant time, if we might have to reallocate.

**Inserting and deleting in a Linked List are constant-time operations if we already know where to insert/delete. But the Table delete operation includes finding the item, and in a sorted list, the Table insert operation needs to find the correct spot. These "find" operations require linear time.

***Of course, we do not (yet!) know any way to guarantee the tree will *stay* balanced, unless we can restrict ourselves to read-only operations (no insert, delete).

Review
Introduction to Tables [3/4]

Tables can be implemented in many ways.

- Different implementations are appropriate in different circumstances.

In special situations, the (amortized) constant-time insertion for an unsorted array and the logarithmic-time retrieve for a sorted array can be combined!

- Insert all data into an unsorted array, sort the array, then use Binary Search to retrieve data.
- This is a good way to handle Table data with **separate filling & searching phases** (and few or no deletes).

Review
Introduction to Tables [4/4]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant (?)	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

Idea #1: Restricted Table

- Perhaps we can do better if we do not implement a Table in its full generality.

Idea #2: Keep a Tree Balanced

- Balanced Binary Search Trees look good, but how to keep them balanced efficiently?

Idea #3: "Magic Functions"

- Use an unsorted array of key-data pairs. Allow array items to be marked as "empty".
- Have a "magic function" that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:

- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables

Priority Queues
Introduction [1/2]

Our next ADT is **Priority Queue**.

- It has almost the same operations as Queue.
- The difference is that each data item has a key, called its **priority**. The item retrieved/deleted is the item with the highest priority.

Conceptually, a Priority Queue is another way to do "standing in line".

- However, items are not handled in the order they were inserted, but rather in order of priority.

Thus, a Priority Queue is not a Queue!

Priority Queues
Introduction [2/2]

We have discussed turning a Sequence into a Queue.

- In a Sequence, we retrieve/delete at **any given position**.
- In a Queue, we can retrieve/delete only the element at the **highest position**.

We can similarly turn a (sorted) Table into a Priority Queue.

- In a Table, we retrieve/delete the item with **any given key**.
- In a Priority Queue, we can retrieve/delete only the element with the **highest key** (priority).

Thus, a Priority Queue is a restricted-access (sorted) Table, just as a Queue is a restricted-access Sequence.

Priority Queues
Definition

Priority Queue has the following data and operations. (These differ a little from those in the text.)

- Data
  - A collection of items, each of which has a priority.
- Operations
  - The Usual
    - **create**, **destroy**, **copy**.
    - **isEmpty**.
  - Operations Specific to Priority Queues
    - **insert**. Given an item (which includes a priority).
    - **getFront**. Gets item with highest priority.
    - **delete**. Removes item with highest priority.

Priority Queues
Usage

A PQ is useful when we have items to process and some are clearly more important than others.

- If an item’s priority includes a time stamp, then we can simulate a mixed Queue/PQ.
  - Items with the same priority can be dealt with in FIFO order.
- Or a mixed Stack/PQ, if you like.
  - What is this good for? I couldn’t say.

PQs are also useful for solving the General Sorting Problem.

- Insert all items, then retrieve/delete all items. The resulting sequence is sorted by priority.
- We can also use a PQ to handle variations on the General Sorting Problem, in which a sequence is modified in the middle of a sort.

Priority Queues
Implementation

We can implement a Priority Queue using any of the methods we have discussed for implementing a Table.

- And they are all still just as dissatisfying. ☹
- In particular, they all have linear-time delete.

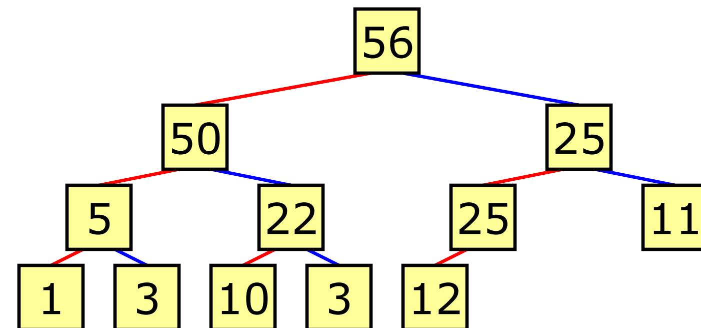
The most interesting thing about PQs is the way they are *usually* implemented, using a data structure called a "Heap".

- We discuss this next …

Heap Algorithms
Definition

We define a **Heap** to be a complete Binary Tree that

- Is empty,
- Or else
  - The root's key (priority) is ≥ than the key of each of the root's children, if any, and
  - Each of the root's subtrees is a Heap.



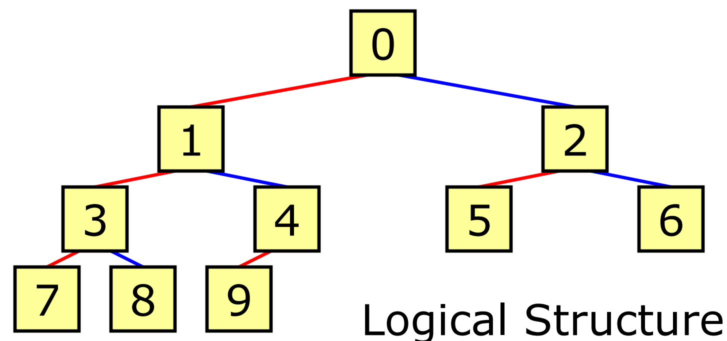
Notes

- This is a **Maxheap**. If we reverse the order, so that the root's key is ≤ than the keys of its children, we get a **Minheap**.
- The text presents Heap as an ADT with essentially the same operations as a Priority Queue. I am not doing this.
- As we will see, a Heap is a good basis for an implementation of a Priority Queue.

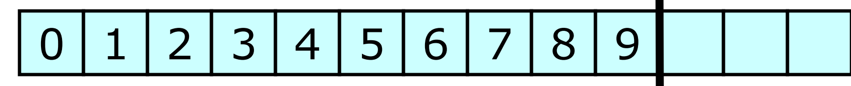
Heap Algorithms
Review — Binary Trees (Implementation)

We discussed an array implementation for **complete** Binary Trees:

- Put the nodes in an array, in the order in which they would be added to a complete Binary Tree.
- No pointers/arrows/indices are required.
- We store *only* the **array** of data items and the **number** of nodes.



Logical Structure

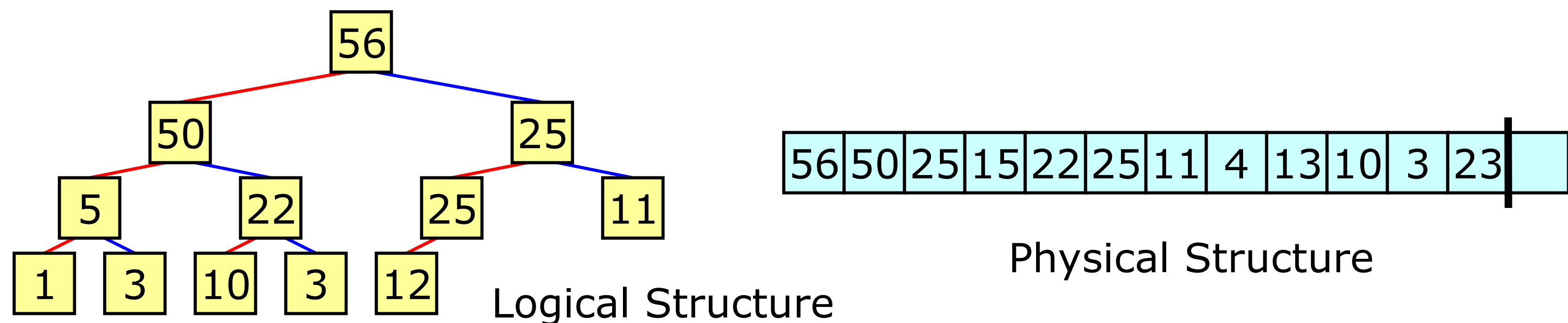


Physical Structure

- Put the root, if any, at index 0.
- The left child of node k is at index $2k + 1$. It exists if $2k + 1 < size$.
- The right child is similar, but at $2k + 2$.
- The parent of node k is at index $(k - 1)/2$ [int division]. It exists if $k > 0$.

Heap Algorithms
Implementation — Data

The usual implementation of a Heap uses this array-based complete Binary Tree.



There are no required order relationships between siblings.

None of the standard traversals gives any sensible ordering.

In practice, we often use "Heap" to mean the array representation.

In order to base a Priority Queue on a Heap, we need to know how to implement the insert & delete operations.

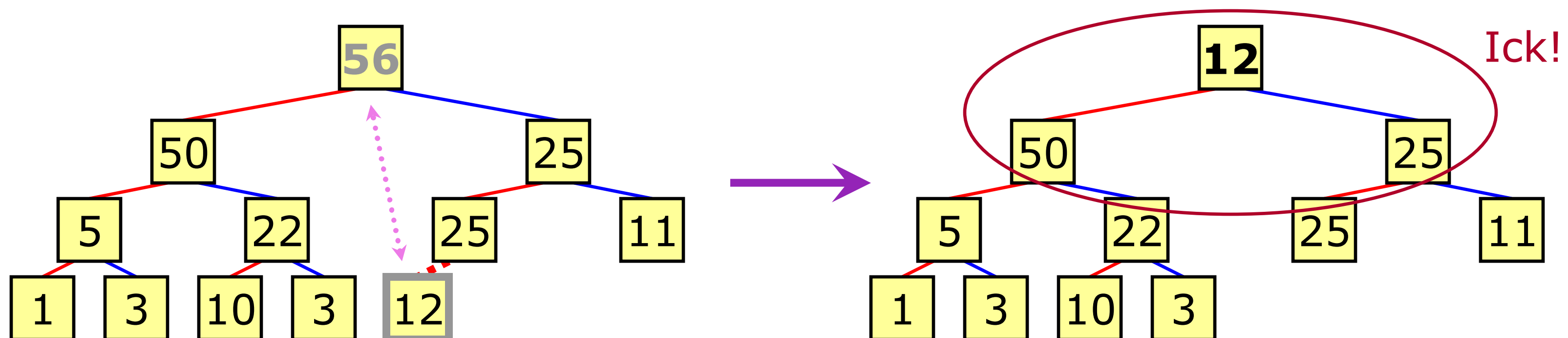
- We look at this next …

Heap Algorithms
Implementation — Delete [1/2]

In a Priority Queue, we can delete the highest-priority item.

In a Maxheap, this corresponds to the root. How do we delete the **root item**, while maintaining the Heap properties?

- We cannot delete the **root node** (unless it is the only node).
- The Heap will have one less item, and so the **last node** must go away.
- But the **last item** is not going away.
- Solution: Move the last node's item to the root; delete the last node.
  - We do this by swapping the items (which has other advantages, as we will see).

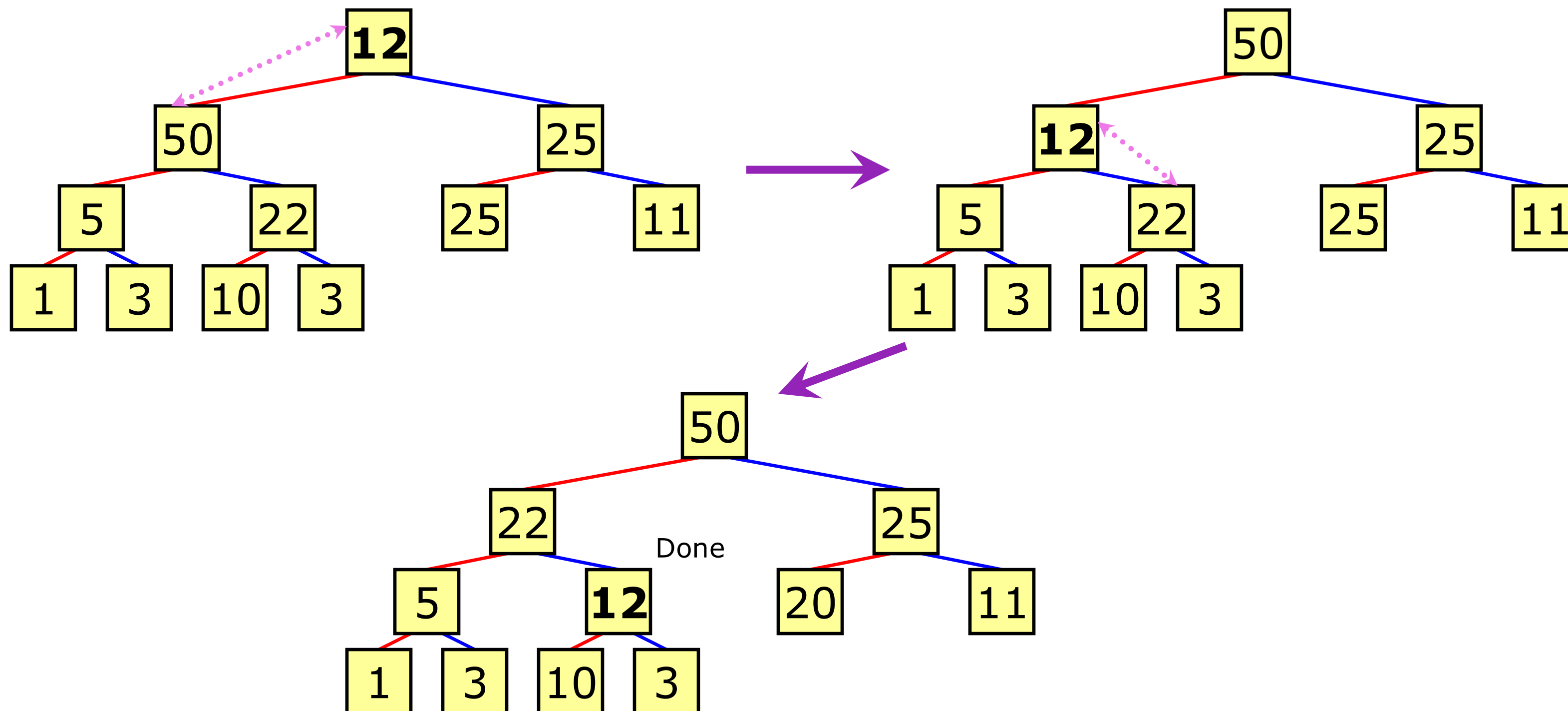


- But now we have another problem: This is no longer a Heap.
  - How do we fix it?

Heap Algorithms
Implementation — Delete [2/2]

To fix: "Trickle down" the new root item.

- If the root is ≥ all its children, stop.
- Otherwise, swap the root item with its **largest** child and recursively fix the proper subtree. (Why largest?)



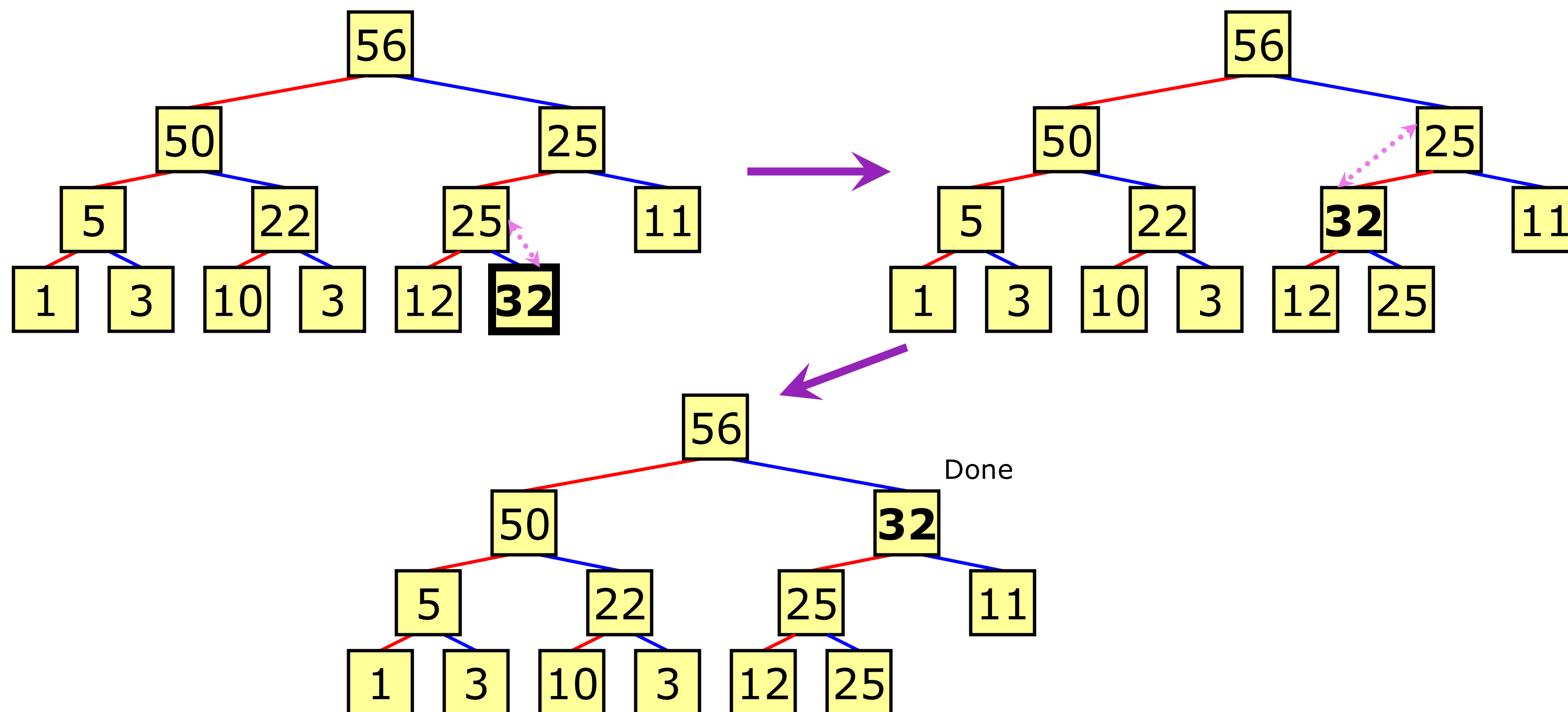
Heap Algorithms
Implementation — Insert

To insert into a Heap, add a new node at the end.

- But if we put our new value in this node, then we may not have a Heap.

Solution: "Trickle up".

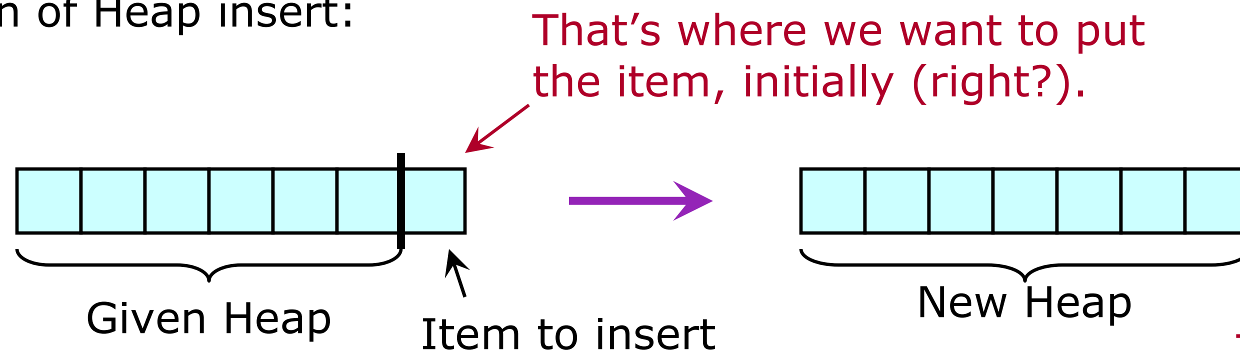
- If new value is greater than its parent, swap them. Repeat at new position.



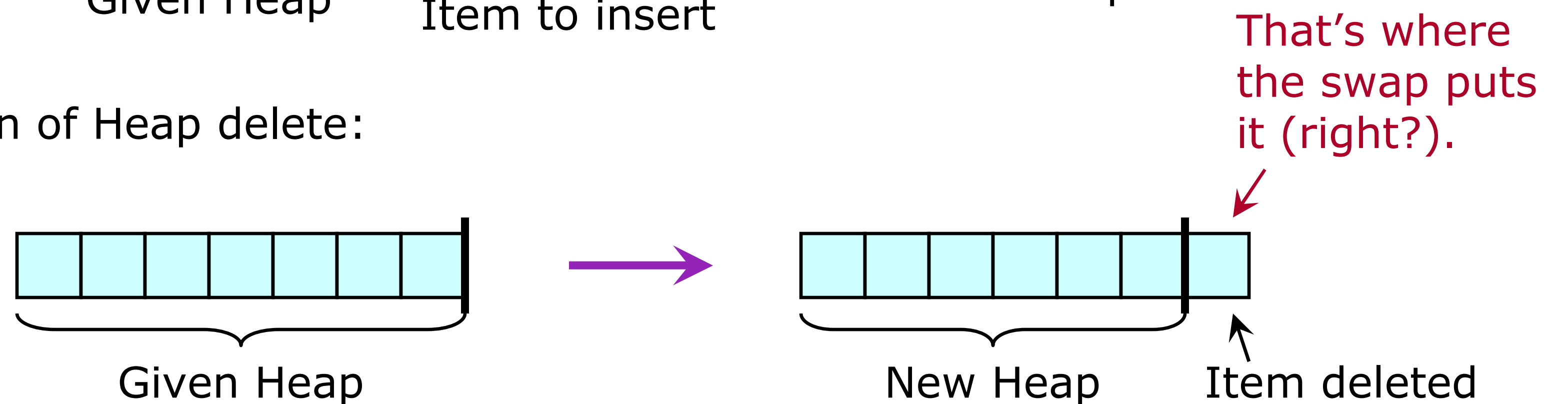
Heap Algorithms
Implementation — Using an Array

Heap insert and delete are usually given a random-access range. The item to insert or delete is last item of the range; the rest is a Heap.

- Action of Heap insert:



- Action of Heap delete:



Note that Heap algorithms can do **all** their work using **swap**.

- This usually allows for both speed and safety.

Heap Algorithms
Efficiency

What is the order of the three main Priority Queue operations, if we use a Heap implementation based on a complete Binary Tree stored in an array?

- **getFront**
  - Constant time.
- **insert**
  - Logarithmic time.
    - Assuming no reallocation, that is, assuming the array is large enough to hold the new item. As on the previous slide, the way that Heaps are **used** often guarantees that this is the case.
  - The number of operations is roughly the height of the tree. Since the tree is balanced, the height is $O(\log n)$.
- **delete**
  - Logarithmic time.
  - No reallocation, of course. Other comments as for insert.

We conclude that a Heap is an excellent basis for an implementation of a Priority Queue.

Heap Algorithms
Example

TO DO

- Write the Heap insert algorithm.
  - Prototype is shown below.
  - The item to be inserted is the final item in the given range.
  - All other items should form a Heap already.

Done. See **`heapalgs.cpp`**, *on the web page.*

```
// Requirements on types:
//     RAIter is a random-access iterator type.
template<typename RAIter>
void heapInsert(RAIter first, RAIter last);
```

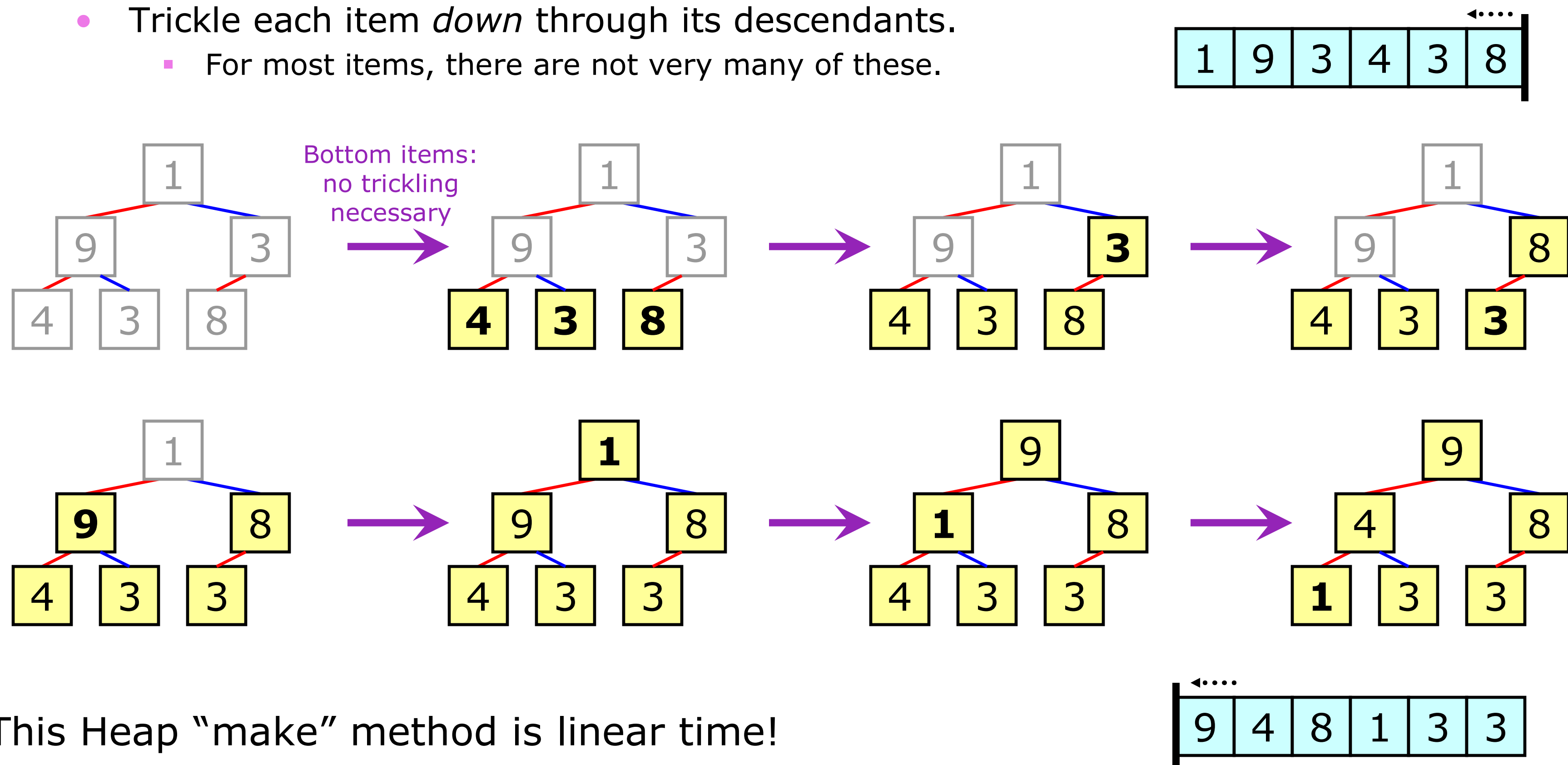
Heap Algorithms
An Efficient "Make" Operation

To turn a random-access range (array?) into a Heap, we *could* do n–1 Heap inserts.

- Each insert operation is $O(\log n)$, and so making a Heap in this way is $O(n \log n)$.

However, we can make a Heap **faster** than this.

- Place each item into a partially-made Heap, in **backwards order**.
- Trickle each item *down* through its descendants.
    - For most items, there are not very many of these.



This Heap "make" method is linear time!

Heap Algorithms
Heap Sort — Introduction

Our last sorting algorithm is **Heap Sort**.

- This is a sort that uses Heap algorithms.
- We can think of it as using a Priority Queue, where the priority of an item is its value.
- Procedure: Make a Heap, then delete all items, using the delete procedure that places the deleted item in the top spot.
- We do a **make** operation, which is $O(n)$, and n getFront/delete operations, each of which is $O(\log n)$.
- Total: $O(n \log n)$.

Heap Algorithms
Heap Sort — Properties

Heap Sort can be done in-place.

- We can create a Heap in a given array.
- As each item is removed from the Heap, put it in the array element that is removed from the Heap.
  - Starting the delete by swapping root and last items does this.
- Results
  - Ascending order, if we used a Maxheap.
  - Only constant additional memory required.
  - Reallocation is avoided.

Heap Sort uses less additional space than Introsort or array Merge Sort.

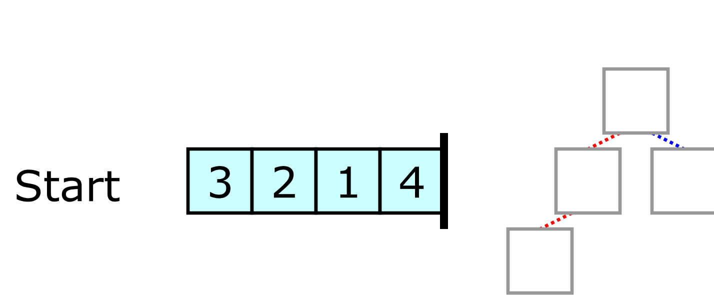
- Heap Sort: $O(1)$.
- Introsort: $O(\log n)$.
- Merge Sort on an array: $O(n)$.

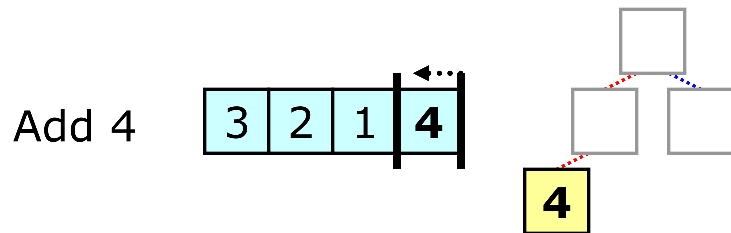
Heap Sort also can easily be generalized.

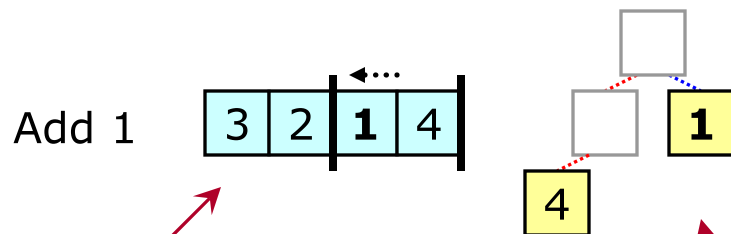
- Doing Heap inserts in the middle of the sort.
- Stopping before the sort is completed.

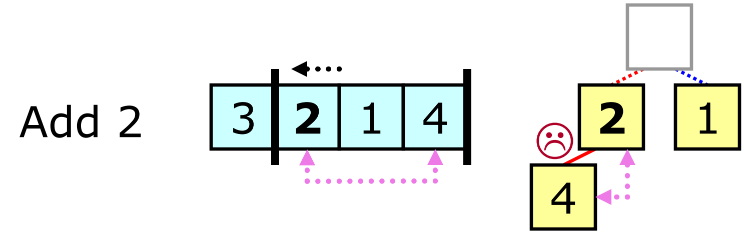
Heap Algorithms
Heap Sort — Illustration [1/2]

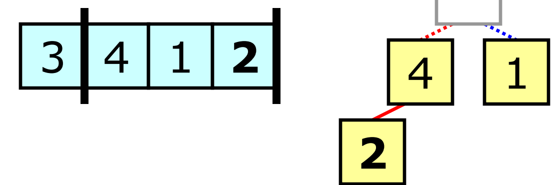
Below: Heap make operation. Next slide: Heap deletion phase.

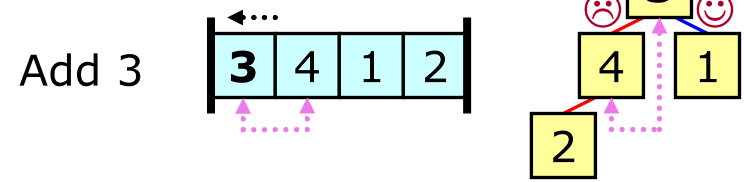


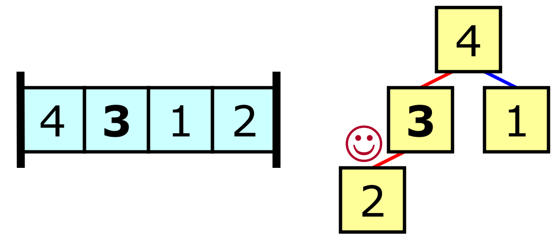










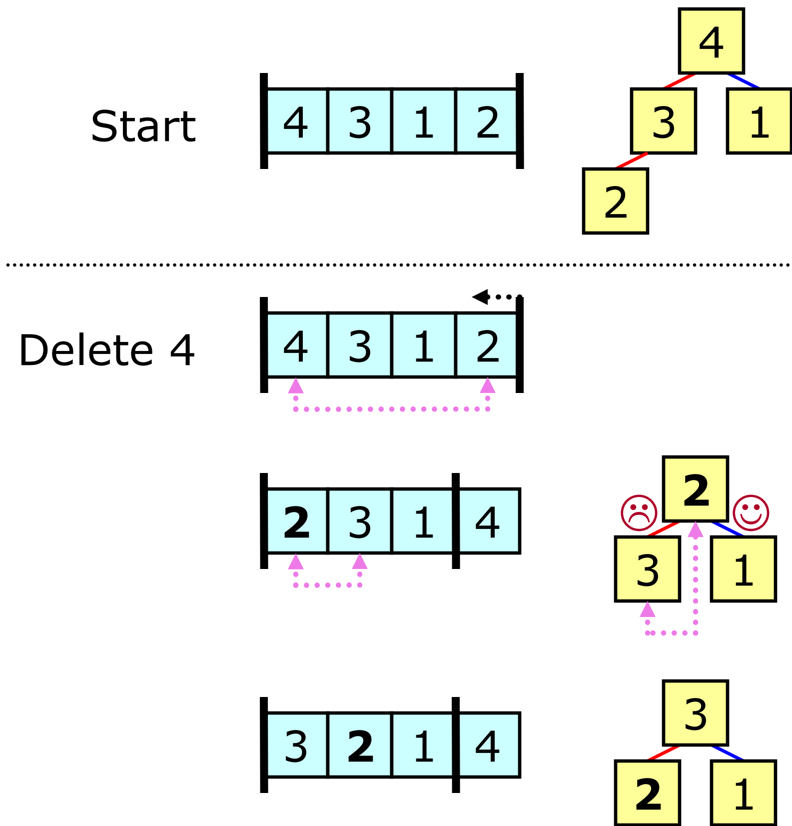


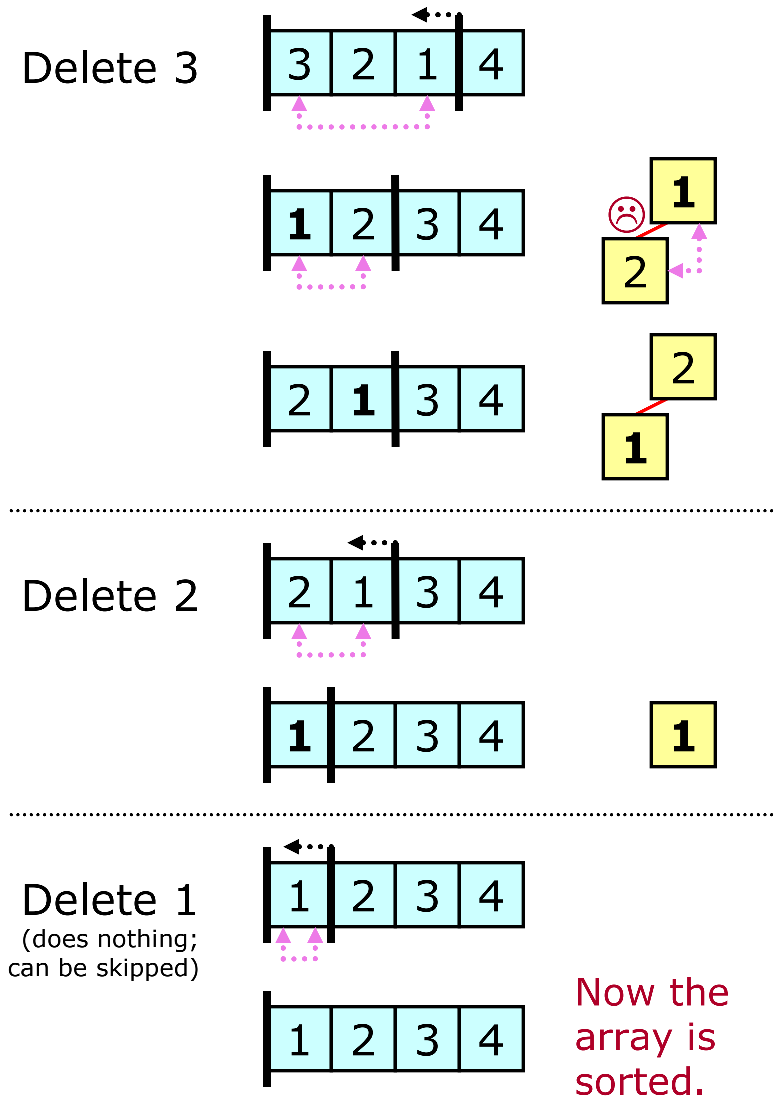
Now the entire array is a Heap.

Note: This is what happens in memory. This is just a picture of the logical structure.

Heap Algorithms
Heap Sort — Illustration [2/2]

Heap deletion phase:





Heap Algorithms
Heap Sort — Analysis

Efficiency ☺

- Heap Sort is $O(n \log n)$.

Requirements on Data ☹

- Heap Sort requires random-access data.

Space Usage ☺

- Heap Sort uses only constant additional storage.
- In particular, it can be done in-place.

Stability ☹

- Heap Sort is not stable.

Performance on Nearly Sorted Data 😐

- Heap Sort is not significantly faster or slower for nearly sorted data.

Notes

- Heap Sort can be generalized to handle sequences that are modified (in certain ways) in the middle of sorting.
- Recall that Heap Sort is used by Introsort, when the recursion depth of Quicksort exceeds the maximum allowed.

Heap Algorithms
Thinking About Heaps

In practice, a Heap is not so much a data structure as it is an ordinary random-access sequence with a particular ordering property.

Associated with Heaps are a collection of algorithms that allow us to efficiently create priority queues and sort data.

- These **algorithms** are the things to remember.
- Thus the subject heading.