Thoughts on Assignment 7
Binary Heap Algorithms continued
Heaps & Priority Queues in the C++ STL

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, November 11, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
The Basics of Trees

Topics

- ✓ Introduction to Trees
- ✓ Binary Trees
- ✓ Binary Search Trees

DONE

Review

Part 1

Review
Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

Note the three primary single-item operations: **retrieve**, **insert**, **delete**. We will see these over & over again.

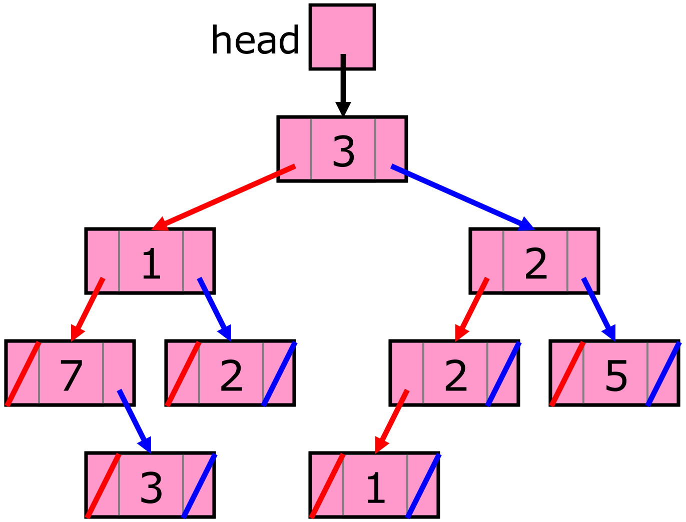
A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Review
Binary Trees — Implementation

A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers—similar to our implementation of a Linked List.

- Each node has a data item and two child pointers: left & right.
- A pointer is null if there is no child.

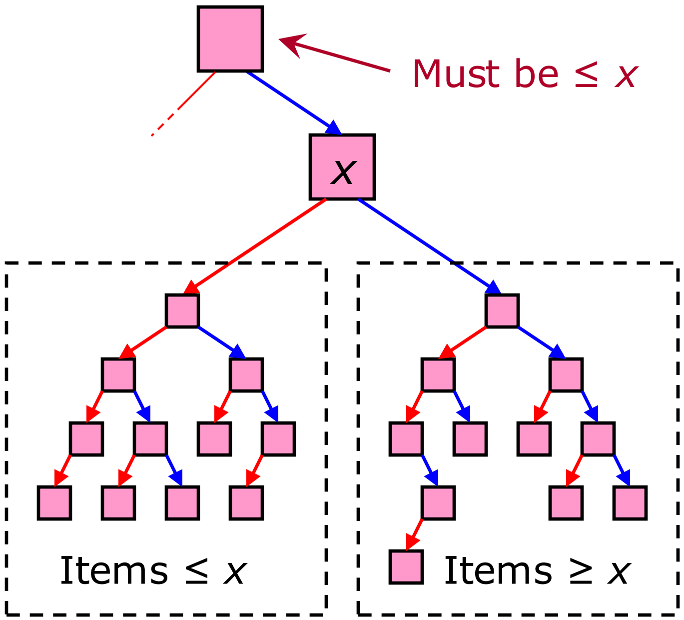


Binary Search Trees — What a Binary Search Tree Is

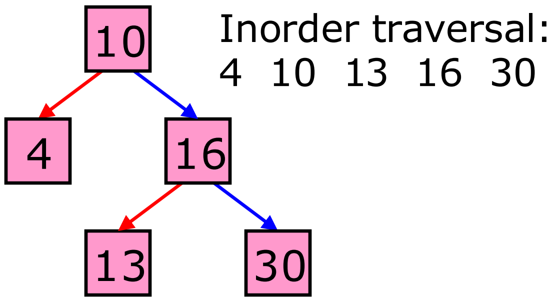
A **Binary Search Tree** is a Binary Tree in which each node contains a single data item, which includes a **key**, and:

- Descendants holding keys less than the node's are in its left subtree.
- Descendants holding keys greater than the node's are in its right subtree.

In other words, an inorder traversal gives keys in sorted order.



In other words, an inorder traversal gives keys in sorted order.



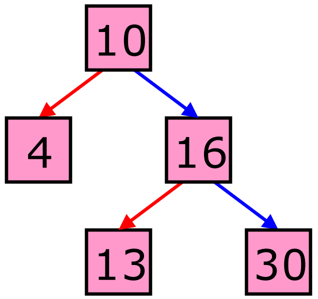
Review
Binary Search Trees — Operations

Algorithms for BST operations:

- Traverse
  - Recursively traverse left subtree of root.
  - Visit the root.
  - Recursively traverse right subtree of root.

Inorder
Traversal

- Retrieve
  - **Search**. Start at the root. Go down, left or right as appropriate, until either the given key or an empty spot is found.
- Insert
  - *Search*, then …
  - Put the value in the spot where it should go.
- Delete
  - *Search*, then …
  - Check the number of children the node has:
    - 0. Delete node.
    - 1. Replace subtree rooted at node with subtree rooted at child.
    - 2. Copy data from (or swap data with) inorder successor. Proceed as above.



Thoughts on Assignment 7

Thoughts on Assignment 7
Overview

In Assignment 7, you will implement the *Treesort* algorithm.

```
template<typename FDIter>
void treesort(FDIter first, FDIter last);
```

Treesort is a general-purpose comparison sort that uses a Binary Search Tree—which you will need to implement.

We look at:

- How Treesort works
- Things you will need to do
  - Writing a Binary Search Tree
  - Finding the value type of an iterator
  - Inserting into a Binary Search Tree
  - Inorder traversal of a Binary Search Tree

Thoughts on Assignment 7
Treesort — Introduction

A sorted container often gives us a sorting algorithm: insert all items into the container, and then iterate through it.

For a SortedSequence, this algorithm is *almost* Insertion Sort. (It would be a non-in-place version of Insertion Sort.)

For a Binary Search Tree, the algorithm is called **Treesort**.

- Procedure
  - Go through the list to be sorted, inserting each item into a BST.
  - Traverse (inorder) the tree; copy each item back to the original list.
- We must allow multiple equivalent keys in our BST.

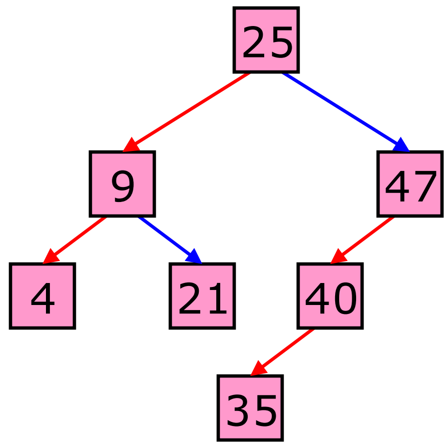
Treesort is not a terribly good algorithm.

Thoughts on Assignment 7
Treesort — Illustration

Treesort this list.

25	47	9	21	40	4	35

- Go through the list to be sorted, inserting each item into a BST.



- Traverse (inorder) the tree; copy each item back to the original list.

4	9	21	25	35	40	47

Thoughts on Assignment 7
Treesort — Properties

What is the order of Treesort?

- Treesort does n BST inserts; each is $\Theta(n)$. Then a traverse: $\Theta(n)$.
- So Treesort is $\Theta(n^2)$. ☹
- However, BST insert is $\Theta(\log n)$ on *average*.
- So Treesort is pretty fast on average: $\Theta(n \log n)$. ☺

Q. Have we seen Treesort before?

A. Kind of. It is much like an unoptimized Quicksort—in disguise.

Two important differences:

- Treesort requires a a large additional memory space, because it does not do its work in the original storage.
- Treesort is not limited by the requirement of a fast *in-place* partition algorithm. As a result, it is stable.

Thoughts on Assignment 7
Writing a Binary Search Tree [1/2]

You will need to define a Binary Search Tree node type.

- Define a class/struct, OR
- Use `std::tuple`.

It is *not* necessary to write a tree class.

- A tree can be handled via a (smart?) pointer to its root node.
- However, you may write a tree class if you wish.

Similarly, we have never written a Linked List class.

In the following slides, I assume you use a simple struct for a Binary Search Tree node, and you refer to a node with a `unique_ptr`. But do things differently if you want.

Thoughts on Assignment 7
Writing a Binary Search Tree [2/2]

Your Binary Search Tree implementation does not need to include all BST operations.

Treesort does the following:

1. Insert all items into a Binary Search Tree.
2. Traverse the tree, copying items to the original range.
3. Destroy the tree.

Destruction is automatic, if you use smart pointers.

So you need to be able to:

- Insert an item, and
- Do an inorder traversal.

Thoughts on Assignment 7
Finding the Value Type

Find the type of the values to sort with `std::iterator_traits`.

```
#include <iterator>
// For std::iterator_traits;

template<typename FDIter>
void treesort(FDIter first, FDIter last)
{
    using Value = typename
        std::iterator_traits<FDIter>::value_type;
```

For code that uses `std::iterator_traits`*, see* `merge_sort.cpp`*.*

Then you can do things like this:

My node struct

```
    auto p = std::make_unique<BSTreeNode<Value>>(…
```

Thoughts on Assignment 7
Binary Search Tree Insert [1/2]

Suggestion. Write a Binary Search Tree insert function that takes:

- A smart pointer to a tree node, *by reference*.
- The item to insert.

Operation:

- If the pointer is null, then set it to point to a new node.
- Otherwise, it points to a node. Compare that node's item with the given item. *Recurse* with the node's left- or right-child pointer, as appropriate.

To insert an item into the tree, call the above function with:

- The head pointer for the tree.
- The item to be inserted.

Thoughts on Assignment 7
Binary Search Tree Insert [2/2]

So your insert function would look like the following.

```
template<typename Value>
void insert(unique_ptr<BSTreeNode<Value>> & head,
            const Value & item);
```

Thoughts on Assignment 7
Inorder Traversal

Suggestion. Write an inorder traversal function that takes:

- A smart pointer to a tree node, *by reference-to-const*.
- An iterator, *by reference*.

Operation:

- If the pointer is null, return.
- Recurse with the left-child pointer.
- Write the data to the location referenced by the iterator.
- Increment the iterator.
- Recurse with the right-child pointer.

`*iter++ = unptr->_data;`

To copy the data in the tree back to the original storage, call the above function with:

- The head pointer for the tree.
- Parameter `first` from function `treesort`.

Thoughts on Assignment 7
Final Note

What about stack overflow?

The Binary Search Tree created during Treesort can easily have large height—for example, if a large range is already sorted.

And this can make recursive insertion and traversal algorithms crash due to stack overflow.

However, you are not required to deal with this issue in Assignment 7. In particular, the test program will not test your code with data that will cause stack overflow in a reasonable implementation.

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables
- ✓ Priority Queues
- (part) Binary Heap Algorithms
- Heaps & Priority Queues in the C++ STL
- 2-3 Trees
- Other self-balancing search trees
- Hash Tables
- Prefix Trees
- Tables in the C++ STL & Elsewhere

Review

Part 2

Introduction to Tables [1/2]

A **Table** allows for look-up by arbitrary key.

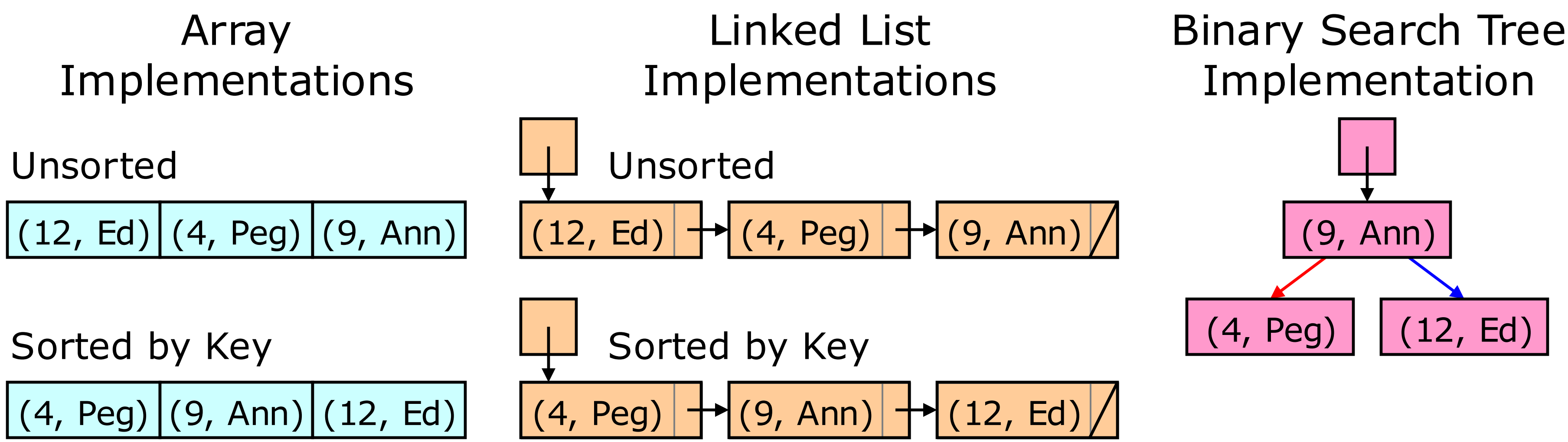
Three primary operations: retrieve, insert, delete.

Possible Table implementations (all too slow!):

- A Sequence holding key-value pairs.
  - Array-based or Linked-List-based.
  - Sorted or unsorted.
- A Binary Search Tree holding key-value pairs.
  - Implemented using a pointer-based Binary Tree.

Table

Key	**Value**
12	Ed
4	Peg
9	Ann



Review
Introduction to Tables [4/4]

Three ideas for dealing with the fact that every Table implementation we can think of is inefficient.

Idea #1: Restricted Tables

- Only allow retrieve/delete on the greatest key.
- In practice: Priority Queues

Idea #2: Keep a tree balanced

- In practice: Self-balancing search trees (2-3 Trees, etc.)

Idea #3: Magic functions

- Use an unsorted array. Each item can be a key-value pair or empty.
- A magic function tells the index where a given key is stored.
- Retrieve/insert/delete in constant time? No, but still a useful idea.
- In practice: Hash Tables

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables ← Several lousy implementations
- ✓ Priority Queues
- (part) Binary Heap Algorithms
- Heaps & Priority Queues in the C++ STL
- 2-3 Trees
- Other self-balancing search trees
- Hash Tables
- Prefix Trees ← A special-purpose implementation: "the Radix Sort of Table implementations"
- Tables in the C++ STL & Elsewhere

Idea #1: Restricted Table

Idea #2: Keep a tree balanced

Idea #3: Magic functions

Priority Queues

A **Priority Queue** is a restricted-access Table. We can insert any item, but we only retrieve/delete the item with the greatest key.

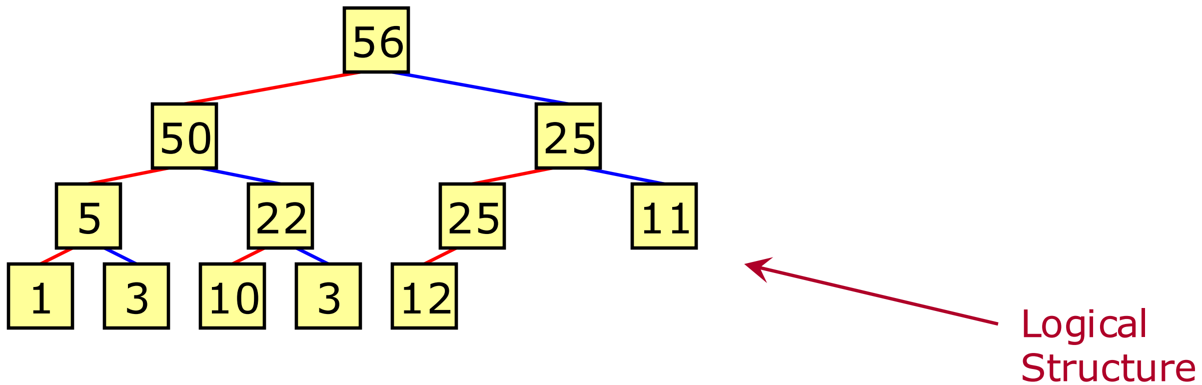
Despite the name, a Priority Queue is *not* a Queue!

We will see an application of a Priority Queue later in the semester.

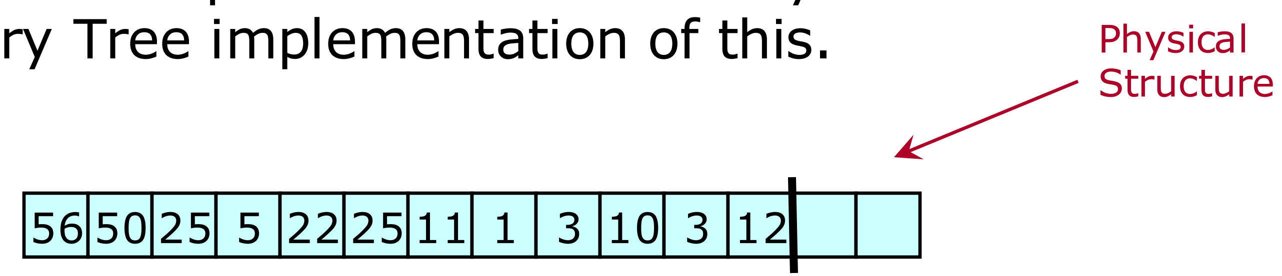
The most interesting thing about Priority Queues is their most common implementation: a structure called a *Binary Heap*.

Review
Binary Heap Algorithms [1/8]

A **Binary Heap** (or just **Heap**) is a complete Binary Tree with one data item—which includes a **key**—in each node, where no node has a key that is less than the key in either of its children.

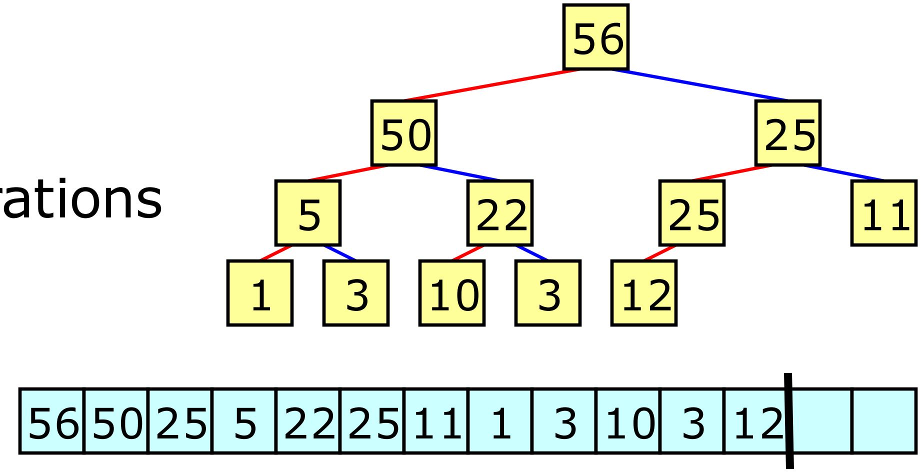


In practice, we use "Heap" to refer to the array-based complete Binary Tree implementation of this.



Binary Heap Algorithms [2/8]

A Heap is a good basis for an implementation of a Priority Queue.



Algorithms for three primary operations

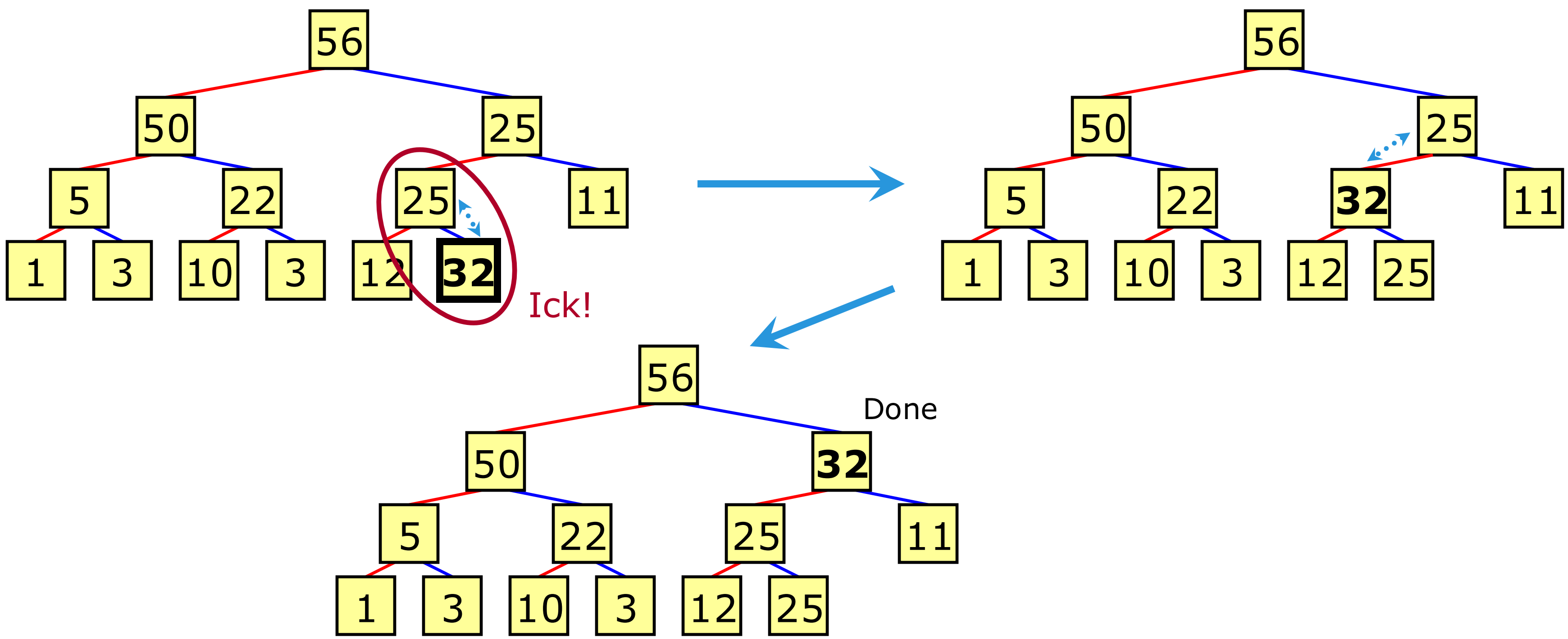
- **getFront**
  - Get the root node.
  - Constant time.
- **insert**
  - Add new node to end of Heap. Sift-up last item.
  - Logarithmic time if no reallocation required.
  - Linear time otherwise. However, in practice, a Heap often does not manage its own memory, which makes the operation logarithmic time.
- **delete**
  - Swap first & last items. Reduce size of Heap. Sift-down new root item.
  - Logarithmic time.

Faster than linear time!

Review
Binary Heap Algorithms [3/8]

To insert into a Heap, add new node/item at end. Then **sift-up**.

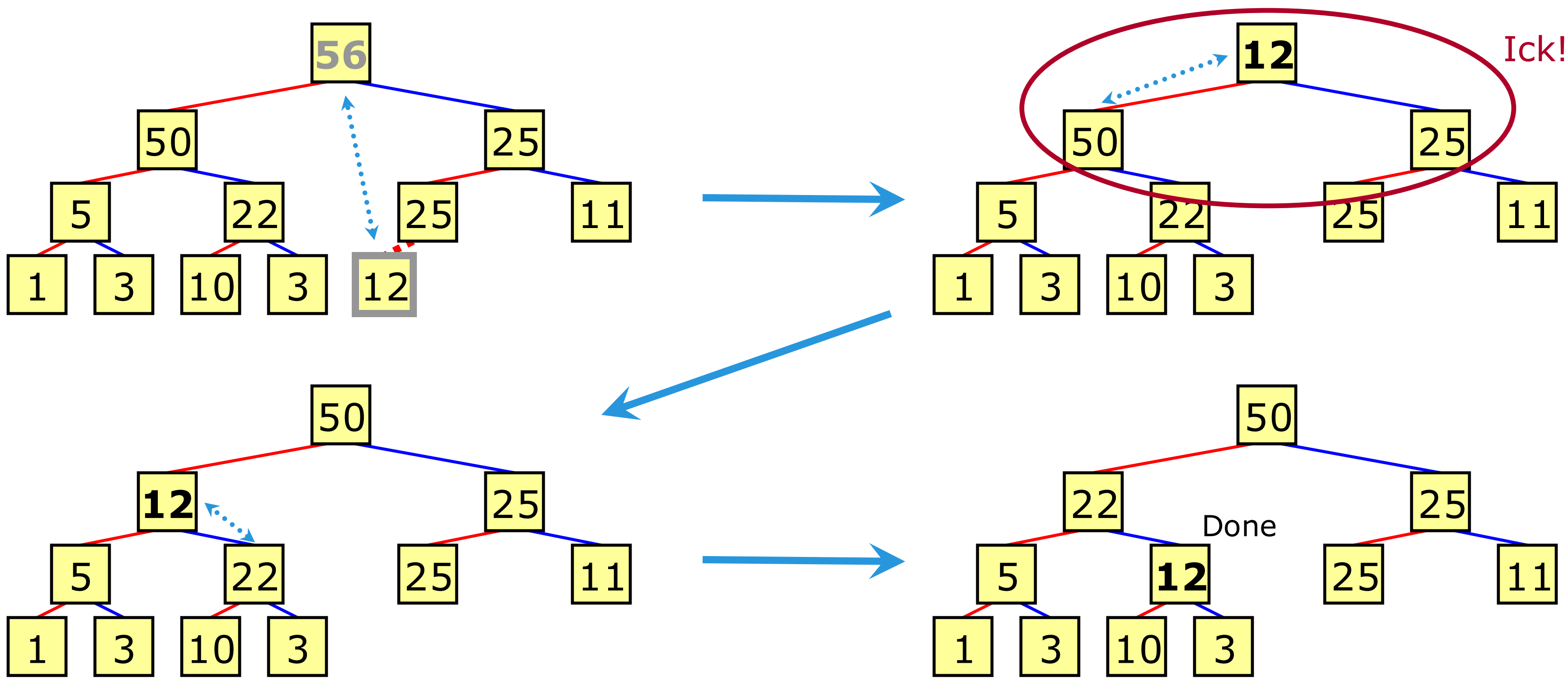
- If value is in root, or is ≤ its parent, then stop.
- Otherwise, swap item with parent. Repeat at new position.



Review
Binary Heap Algorithms [4/8]

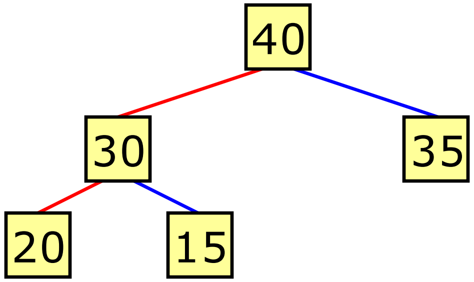
To delete the root item from a Heap, swap root & last items, and reduce the size of the Heap. The **sift-down** the new root item.

- If value is ≥ all of its children, the stop.
- Otherwise, swap item with its *largest* child. Repeat at new position.



Review
Binary Heap Algorithms [5/8] (Try It!)

Do Heap insert 50 on the Binary Heap shown below. Draw the resulting Heap, as a tree.

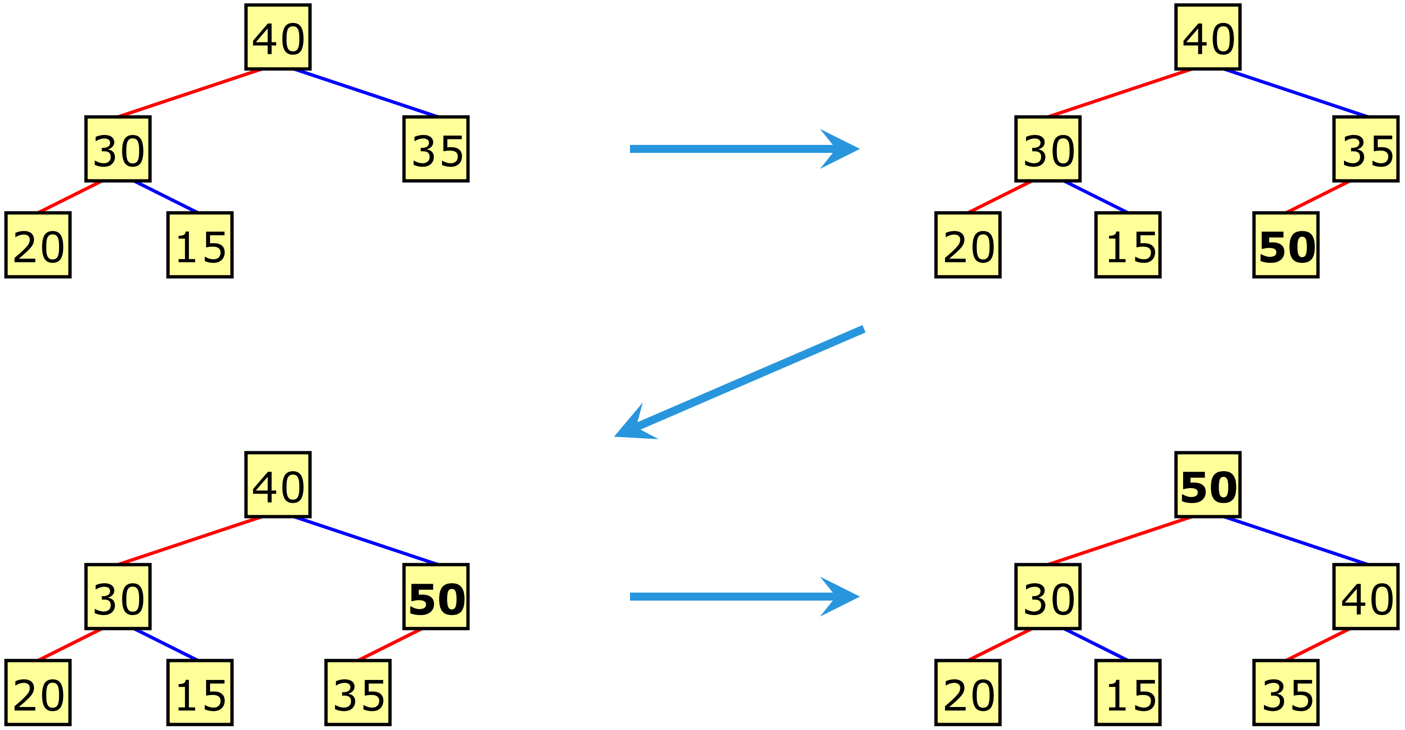


Answer on next slide.

Binary Heap Algorithms [6/8] (Try It!)

Do Heap insert 50 on the Binary Heap shown below. Draw the resulting Heap, as a tree.

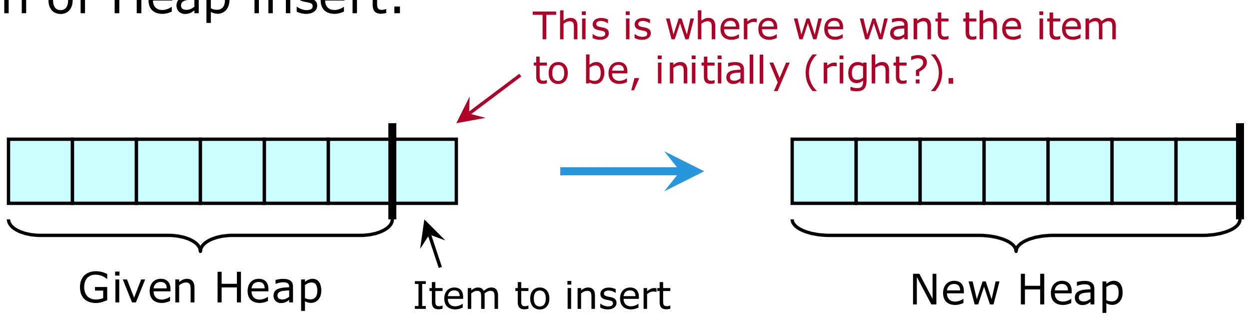
Answer



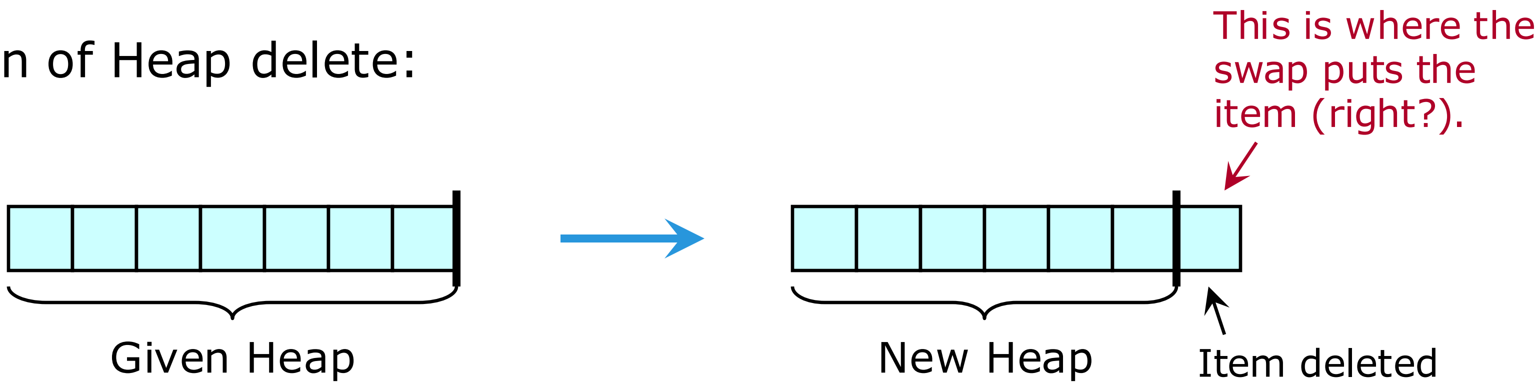
Review
Binary Heap Algorithms [7/8]

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

- Action of Heap insert:



- Action of Heap delete:



Note that Heap algorithms can do *all* modifications using *swap*.
This usually allows for both speed and (exception) safety.

Review
Binary Heap Algorithms [8/8]

DONE

- 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.
- Write other Heap algorithms as time permits.

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

See `heap_algs.hpp`*. The other Heap algorithms have also been written.*

See `heap_algs_main.cpp` *for a program that uses this header.*

Binary Heap Algorithms

continued

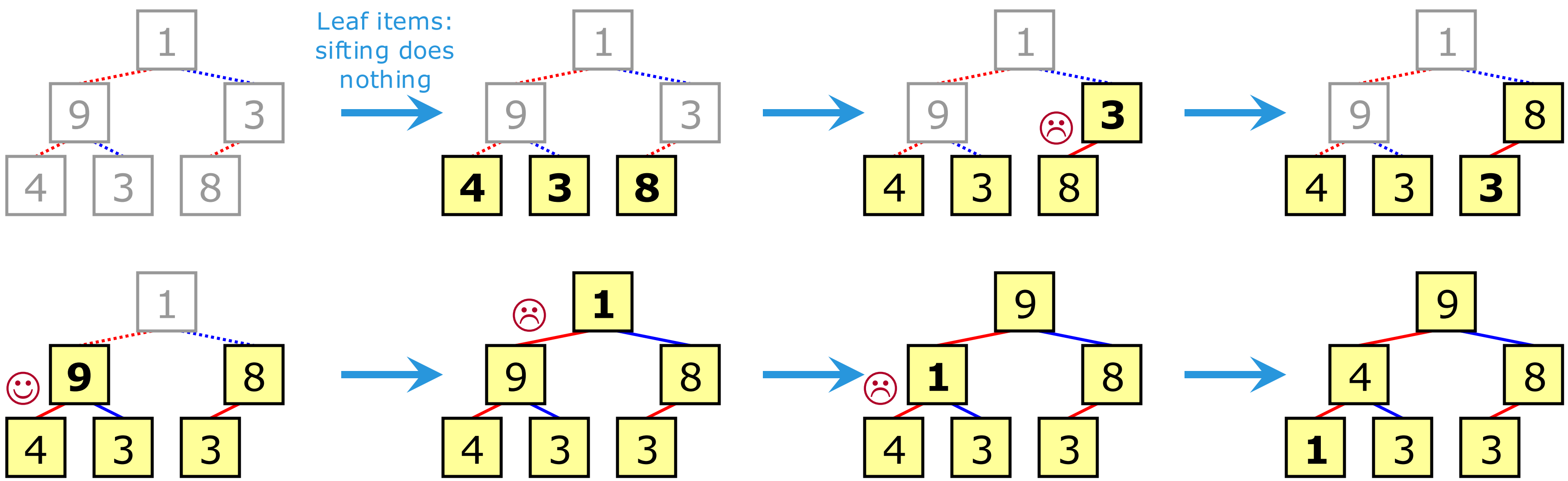
Binary Heap Algorithms
Fast Make-Heap

To turn a range into a Heap, we could do $n-1$ Heap inserts. Each insert op is $\Theta(\log n)$; making a Heap in this way is $\Theta(n \log n)$.

However, there is a faster way.

- Go through the items in reverse order.
- Sift-down each item through its descendants.

1	9	3	4	3	8



This Make-Heap algorithm is *linear time*!

9	4	8	1	3	3

Heap Sort — Introduction

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

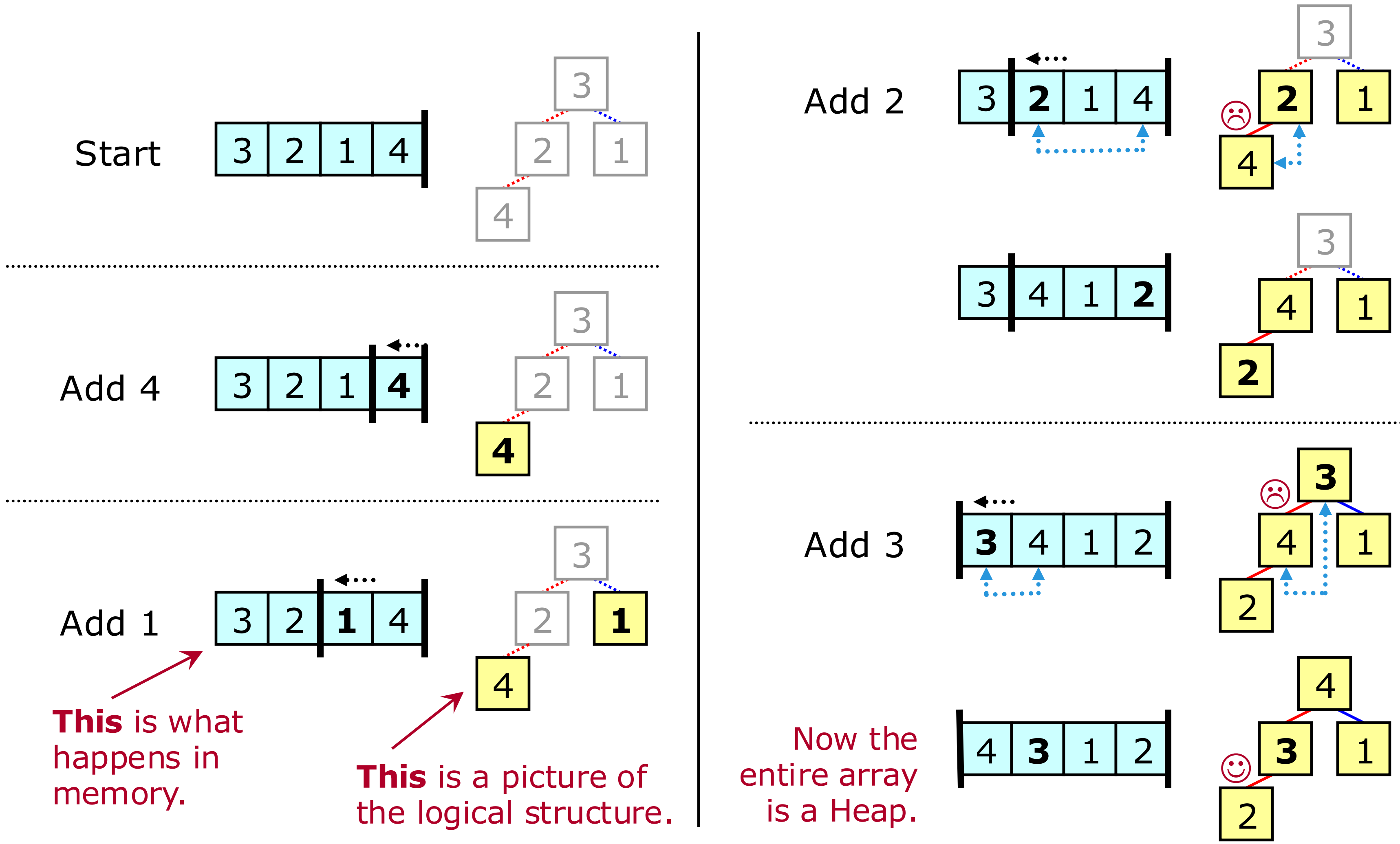
- This is a fast comparison sort that uses Heap algorithms.
- We can think of it as using a Priority Queue, except that the algorithm is in-place, with no separate data structure used.
- Procedure. Make a Heap, then delete all items, using the Heap delete procedure that places the deleted item in the top spot.

Is Heap Sort efficient?

- The *Make-Heap* operation is $\Theta(n)$.
- Then we do *n Heap delete* operations, each of which is $\Theta(\log n)$.
- Total: $\Theta(n \log n)$.

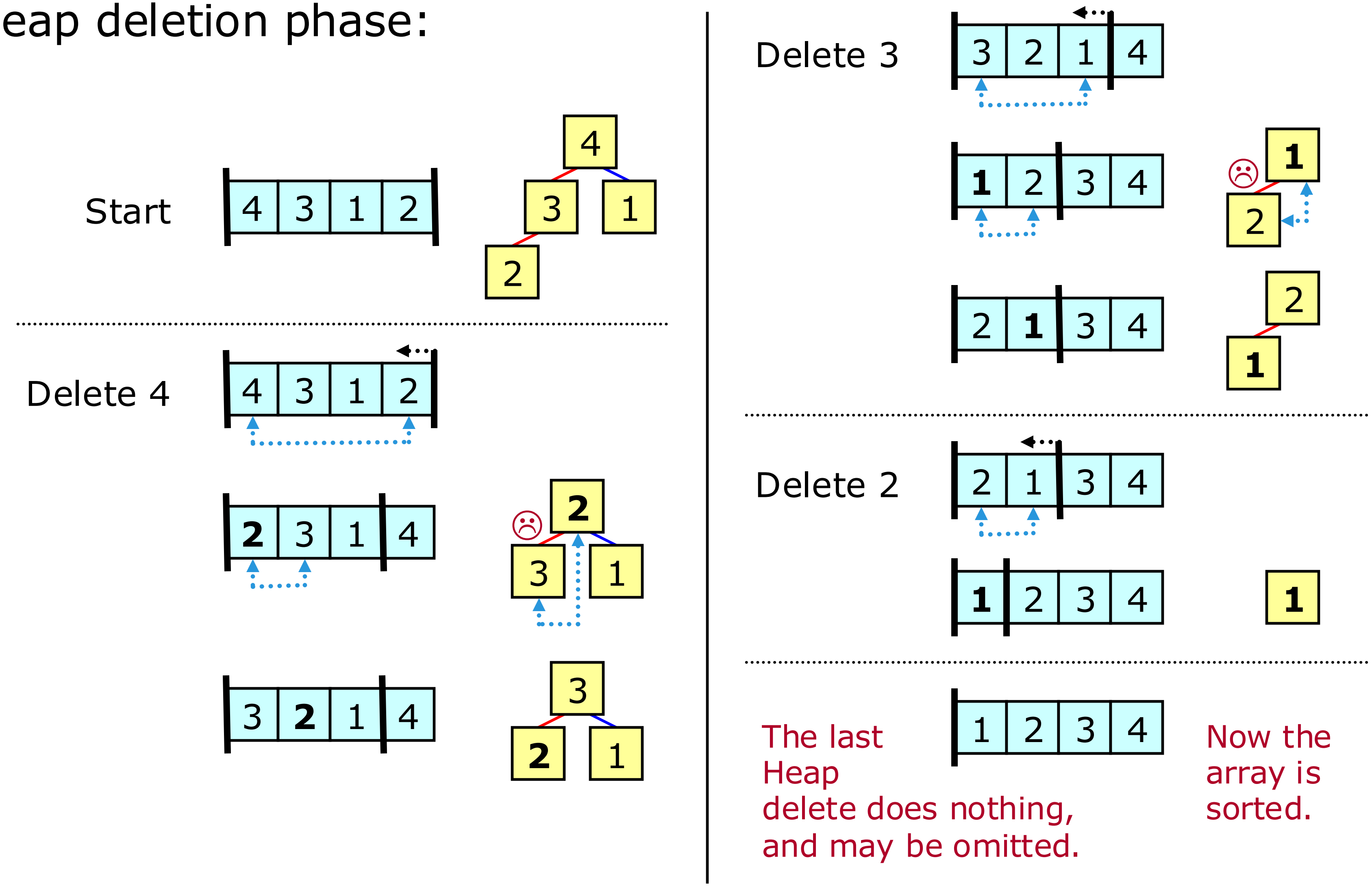
Binary Heap Algorithms
Heap Sort — Illustration [1/2]

Below: Make-Heap operation. Next slide: Heap deletion phase.



Binary Heap Algorithms
Heap Sort — Illustration [2/2]

Heap deletion phase:



Binary Heap Algorithms
Heap Sort — Properties

Heap Sort is in-place.

- We can create a Heap in a given array.
- As each item is removed from the Heap, put it in the array item that was 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 is required. No reallocation is done.

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

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

Heap Sort can easily be generalized.

- Stopping before the sort is finished.
- Doing Heap insert operations in the middle of the sort.

Binary Heap Algorithms
Heap Sort — CODE

TO DO

- Write Heap Sort, using the Heap algorithms that are already written.

Done. See `heap_sort.cpp`*.*

Binary Heap Algorithms
Heap Sort — Analysis

Efficiency ☺

- Heap Sort is Θ(n log n).

Requirements on Data ☹

- Heap Sort requires random-access data.

Space Usage ☺

- Heap Sort is 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.

We have seen these together before (Iterative Merge Sort on a Linked List), but never for an array.

Notes

- Heap Sort is significantly slower than both Merge Sort and Introsort.
- Heap Sort can be stopped early, with useful results.
- Recall that Heap Sort is the usual fallback algorithm in Introsort.

Binary Heap Algorithms
Final Thoughts

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

Associated with Heaps are a collection of algorithms that allow us to efficiently create Priority Queues and do comparison sorting.

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

Heaps & Priority Queues in the C++ STL

Heaps & Priority Queues in the C++ STL
Heap Algorithms [1/2]

The C++ STL includes several Heap algorithms, in `<algorithm>`.

- Each takes a range specified by a pair of random-access iterators.
- An optional third parameter is a custom comparison.

These are the same algorithms implemented in `heap_algs.hpp`*.*

`std::push_heap`

- Heap insert. [*first*, *last*–`1`) is a Heap. Item to insert is `*(`*last*–`1)`.

`std::pop_heap`

- Heap delete. Puts the deleted element in `*(`*last*–`1)`.

`std::make_heap`

- Make a range into a Heap, using the fast Make-Heap algorithm.

`std::sort_heap`

- Given a Heap. Does n–1 `pop_heap` calls.
  Result: sorted range. (So `make_heap` + `sort_heap` does Heap Sort.)

`std::is_heap`

- Test whether a range is a Heap. Returns `bool`.

Heaps & Priority Queues in the C++ STL
Heap Algorithms [2/2]

`std::partial_sort`, in `<algorithm>`, does Heap Sort, but may stop early.

- It takes three iterators: *first*, *middle*, *last*, and an optional comparison.
- [*first*, *last*) must be a valid range, and *middle* must be between *first* & *last* (inclusive).
- It does Heap Sort—in reverse order using a Minheap—stopping when the range [*first*, *middle*) has been filled with sorted data.
- Result. The range [*first*, *middle*) contains exactly the data it would if the entire range were sorted. The range [*middle*, *last*) contains the items it would contain if the entire range were sorted, but they may not be in sorted order.

A variation, `std::partial_sort_copy`, also in `<algorithm>`, leaves data in the original range unchanged, placing its results in a second provided range.

std::priority_queue — Introduction

Note the name of the header!

The STL has a Priority Queue: `std::priority_queue` (`<queue>`).

This is another *container adapter*: wrapper around a container. And once again, you get to pick what that container is.

```
std::priority_queue<T, container<T>>
```

- `T` is the value type.
- *container*`<T>` can be any standard-conforming *random-access* sequence container with value type `T`.
- In particular, *container* can be `vector` or `deque`.
- But not `list`, as it is not random-access.

container defaults to `std::vector`.

```
std::priority_queue<T>
    // = std::priority_queue<T, std::vector<T>>
```

Heaps & Priority Queues in the C++ STL
std::priority_queue — Members

The member function names used by `std::priority_queue` are the same as those used by `std::stack`.

- Not those used by `std::queue`.
- So `std::priority_queue` has `top`, not `front`.

Given a variable `pq` of type `std::priority_queue<T>`, we can do:

- `pq.top()`
- `pq.push(`*item*`)`
    - *item* is some value of type `T`.
- `pq.pop()`
- `pq.empty()`
- `pq.size()`

Heads & Priority Queues in the C++ STL
std::priority_queue — Comparison [1/2]

The comparison used by `priority_queue` defaults to `operator<`.

```
priority_queue<Foo> pq1;  // Use operator<
```

We can specify a custom comparison. An optional template parameter is the *type* of a comparison object.

Below is a Priority Queue whose top value is the smallest.

We give the third template argument, so we must also give the second.

```
priority_queue<Foo, std::vector<Foo>,
                  std::greater<Foo>> pq2;  // Use operator>
```

The *type* of the comparison

If, as above, we pass no ctor arguments, then our comparison object is a default-constructed object of the given type.

Heaps & Priority Queues in the C++ STL
std::priority_queue — Comparison [2/2]

To pass our own comparison function, we specify:

- Its type, as the third template argument.
- The comparison itself, as a constructor argument.

```
auto comp = [](const Foo & a, const Foo & b)
{ return a.bar() < b.bar(); };
```

Definition of our comparison

```
std::priority_queue<Foo, std::vector<Foo>,
                    decltype(comp)> pq3(comp);
```

The *type* of our comparison

Our comparison

See `pq.cpp` *for example code.*

We will see a more practical example near the end of the semester.

Overview of Advanced Table Implementations

This ends our coverage of Idea #1: restricted Tables.

Next, actual Tables, allowing retrieve & delete for arbitrary keys.

We cover the following advanced Table implementations.

- Self-balancing search trees
  - To make things easier, allow more children (?):
    - **2-3 Tree**
      - Up to 3 children
    - **2-3-4 Tree**
      - Up to 4 children
    - **Red-Black Tree**
      - Binary Tree representation of a 2-3-4 Tree
  - Or back up and try for a strongly balanced Binary Search Tree again:
    - **AVL Tree**
- Alternatively, forget about trees entirely:
  - **Hash Table**
- Finally, "the Radix Sort of Table implementations":
  - **Prefix Tree**

Idea #2:
Keep a tree balanced

Later, we cover other self-balancing search trees: B-Trees, B+ Trees.

Idea #3:
Magic functions