“Abstract Data Type”

- Authors mildly disagree over the following definitions, but we will go with Shaffer's version (read §1.2).

- In CS, a *data type* is a *collection of values* . . .

- integer operations: ?, string operations: ?
“Abstract Data Type”

- Authors mildly disagree over the following definitions, but we will go with Shaffer's version (read §1.2).

- In CS, a data type is a collection of values . . .

  - (Standard) integer, real number, string of characters, reference.

  - integer operations: . . .

  - string operations: . . .
“Abstract Data Type”

- Authors mildly disagree over the following definitions, but we will go with Shaffer's version (read §1.2).

- In CS, a *data type* is a *collection of values* . . .
  - (Standard) integer, real number, string of characters, reference.
  - integer operations: +, etc., string operations: +, etc., array ops: [i] etc.
  - Aggregate data types give access to the individual pieces.

- An *abstract data type* is "the specification of a data type within some language, independent of an implementation." The last part is the important part: we want to talk about what a data type will store, and what it can do, without being tied down to how.
“Abstract Data Type”

- Authors mildly disagree over the following definitions, but we will go with Shaffer's version (read §1.2).

- In CS, a *data type* is a *collection of values* . . .
  - (Standard) integer, real number, string of characters, reference.
  - (Custom, aggregate) node of a linked list, employee record, array of employee records, linked list of employee records.

- . . . plus a *set of operations* on those values.
  - integer operations: +, etc., string operations: +, etc., array ops: [i] etc.
Abstract Data Type - Examples

• In Java, the *interface* concept is equivalent.
  – ADT is an older and more widespread term – not everyone uses Java.
Abstract Data Type - Examples

- More complex data types:
  - Linked List
  - Sequence (has size, first, current, next, last)
  - Stack
  - Queue (a/k/a FIFO)
  - Hash table
  - Binary search tree
Abstract Data Type - Examples

• More complex data types:
  – Linked List
  – Sequence (has size, first, current, next, last)
  – Stack
  – Queue (a/k/a FIFO)
  – Hash table
  – Binary search tree

• Data structure = ADT plus implementation
  – Complex data types can be implemented with simpler data types, for example, _________
Abstract Data Type - Examples

- **Data structure** = ADT plus implementation
  - Complex data types can be implemented with simpler data types, for example,
    - Linked List from simple aggregate nodes
    - Sequence from Linked List, or from Array
    - Stack from Sequence
    - Queue from Sequence
    - Queue from Array (tricky – see § 4.12.2)
    - Binary tree from simple aggregate nodes
    - Binary tree from Array (also tricky – § 6.16)

- **Key points:** know bold terms, p. 4, for ADT.
Aside: container ADTs

- Stack
- Queue
- Sequence
- Linked List
- Array
Aside: container ADTs

Stack
Queue
Sequence
Linked List
Array

Create
Push
Pop
is_Empty?
Aside: container ADTs

Stack
Queue
Sequence
Linked List
Array

Create
Enqueue
Dequeue
is_Empty?
Aside: container ADTs

- Stack
- Queue
- Sequence
- Linked List
- Array

Create, Size?
First, Last
Previous, Next
Insert, Erase
Aside: container ADTs

- Stack
- Queue
- Sequence
- Array
- Linked List

This arrow means,
You can implement the Queue ADT using a Sequence.
Aside: container ADTs

- Stack (Last in, First out)
- Queue (First in, First out)
- Sequence
- Linked List
- Array

* A crowd boards the elevator at the top floor of a building.

* People going down press buttons for various floors. E.g., 3, 2, 8, 4

* Which floor should the elevator go to next?
  LIFO order?
  FIFO order?
Aside: container ADTs

- Stack: Last in, First out
- Queue: First in, First out
- Priority Queue (max queue): Max key is first out.
- Sequence
- Linked List
- Array
Aside: container ADTs

A priority queue is useful for your . . .

* Elevator
* Hard disk controller
* Event simulator
* and much more!
Aside: container ADTs

A heap is a kind of binary tree, with special shape and order properties.
Aside: container ADTs

A heap can easily be implemented in an array!
Sorting (in general) – §§ 7.1-7.2

• How to sort a sequence of records?

• What is a record?
  – A unit of information,
  – has a key, with a total order relation.
  – optionally has associated satellite data (aggregate)
  – Example: ____________.

• Sorting strategies we have seen so far:
  – ___ Sort and ___ Sort, on ____s of records
Sorting (in general) – §§ 7.1-7.2

- How to sort a sequence of records?
- What is a record?
  - A unit of information,
  - has a key, with a total order relation.
  - optionally has associated satellite data (aggregate)
  - Example: (marathon) race-time and racer name.
- Sorting strategies we have seen so far:
  - Insertion Sort and Bubble Sort, on arrays of records
Sorting (in general)

- **Input:** a collection (often an array) of records
  \[ r_1, r_2, r_3, r_4, \ldots, r_n. \]

- **Output:** a permutation of those same records
  \( (r'_1, r'_2, r'_3, r'_4, \ldots, r'_n) \), such that \( \forall i, 1 \leq i < n \) \( r'_i \leq r'_{i+1} \).

- **Time analysis:** how does time-cost depend on \( n \)?

- **Space analysis:** is it *in-place* or not?
  - “In-place” = only uses \( \Theta(1) \) additional storage.

- **Stability:** what if two records \( r_i, r_j \) have equal keys?
  - “Stable” if their order in the input matches order in output.
Sorting (in general)

Insertion sort and Bubble Sort – §§ 7.3-7.4

- Input and output are arrays.
- Time analysis: $O(n^2)$
- Space analysis: $\Theta(1)$ additional storage for the index variables and copy of record (i.e., in-place).
- Stable?
Heaps and Heapsort

https://www.xkcd.com/835/
Sorting with a priority queue

- Fill the max-queue with your records.
Sorting with a priority queue

- Fill the max-queue with your records.
- Repeatedly call `extractMax()` fill output array.

iteration 1
Sorting with a priority queue

- Fill the max-queue with your records.
- Repeatedly call `extractMax()` fill output array.

\[
\begin{align*}
\text{iteration 1} & : \_\_, \_\_, \_\_, \ldots, 89 \\
\text{iteration 2} & : \_\_, \_\_, \_\_, \ldots, 77, 89 \\
\text{iteration } n & : 3, 8, 11, \ldots, 77, 89
\end{align*}
\]
Sorting with a priority queue

- If you do this with a heap, implemented as an array, then you have Heapsort (Williams, 1964).

- So, what is a heap?
  - Binary tree with an “almost complete” shape.
  - Keys obey an order property: each child key is less than or equal to its parent's key. (max heap)

- Aside: such properties are called invariants.
  - Heap shape invariant, max-heap order invariant.
Heap: “almost complete” binary tree

- A heap is a binary tree with $n$ nodes.
- Possibly complete (i.e., all leaves at same depth), or . . .
- Possibly leaves at two depths, all deeper leaves “to the left.”
- This is the shape invariant.
Heap: “almost complete” binary tree

• A heap is a binary tree with $n$ nodes.
• Possibly complete (i.e., all leaves at same depth), or . . .
• Possibly leaves at two depths, all deeper leaves “to the left.”
• This is the shape invariant.

“To the left”: given two leaves $r, s$, if $r$ is deeper than $s$, then let $a$ be their deepest common ancestor. Node $r$ must be in $a$'s left subtree, and $s$ must be in $a$'s right subtree.
Max-heap order invariant

- For each node $n$ that has parent $p$, \( \text{key}(n) \leq \text{key}(p) \).
- Therefore, the root has the maximum key.
- Siblings have no defined order relationship.
- It is not a binary search tree!
- Examples
Implementing a Heap in an array

- Fill the array indices 1 to $n$ with records.
- Index 1 represents the root of the tree.
- Tree-to-array mapping works like this:

Given node with index $j$,
* what is its left child?
* what is its right child?
* what is its parent?
* is it a leaf?
Implementing a Heap in an array

- Fill the array indices 1 to \( n \) with records.
- Index 1 represents the root of the tree.
- Tree-to-array mapping works like this:

Given node with index \( j \),
* what is its left child? Index \( 2j \).
* what is its right child? Index \( 1+2j \).
* what is its parent? Index \( \lfloor j/2 \rfloor \).
* is it a leaf? Test \( 2j > n \).
Inserting a record into a Heap

• Add as a leaf, and “bubble up.”
  – Add new record $r$ as last element in array (leaf).
  – If parent exists, and has key smaller than $\text{key}(r)$, swap positions.
  – Repeat with $r$ until condition is false.

• Not used in Heapsort: instead, we will perform a “mass insertion” of a whole array of records.

• Useful for priority queues.
extractMax() from a Heap

- Maximum value is stored in the root.
- So, return and remove the root node (chop off its head).
- Now we likely have a problem -- two heaps.
- Promote last leaf $r$ to root, and “sift down.”
  - Compare $r$ with its children.
  - If either is larger, $r$ swaps positions with larger child.
  - Follow $r$ down, repeat with $r$ until condition is false.
Heapsort

- At first, the entire size-$n$ input array is turned into a single heap of size $n$ (details on slide 36ff).
- After first `extractMax()`, heap shrinks to size $n-1$. So location $[n]$ in array is unused! Store maximum value there.
- After next `extractMax()`, heap shrinks again. Fill the next unused location with next-largest value.
- Repeat until the heap is empty (or size 1).
- (So, slide 24-25 is misleading: no unused space.)
Sorting with Heapsort

- Turn array of \( n \) records into a max-heap
- Repeatedly call `extractMax()`: shrink heap, grow sorted sequence from far end of array.

Of course, your program must keep track of the heap's size.
Building a heap from an array

- In general, the input to Heapsort will be an array of records in arbitrary order.

- Although we could insert records one by one into an empty heap, it is faster to do perform a “mass insertion”:
  - Treat the array as if it were already the contents of an almost-complete binary tree (satisfying the heap shape property, but violating the heap order property); then,
  - “sift down” each record, starting from the leaves, working toward the root, until the heap order property is satisfied.
Build-Max-Heap

Start with a (perhaps) unsorted array of values.
Build-Max-Heap

Start with a (perhaps) unsorted array of values.

Consider it like it's already a (bad) heap.

(Why bad?)

This picture shows both the tree and the array, for clarity. Of course, there is only one copy in reality.
Build-Max-Heap

Start fixing the order property, starting from the leaves, working backwards.

Each leaf is a (trivial) max-heap.
Build-Max-Heap

We sift down the node at index 6:

It obviously will need to swap with its (only) child.
When sift down terminates, we have a max-heap at that node.
Build-Max-Heap

We work our way backwards through the array.
We work our way backwards through the array.
We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

We work our way backwards through the array.
Build-Max-Heap

At the end, we have a max-heap.
Performance of Heapsort

- Height $h$ of a heap of $n$ nodes: $h = \Theta(\log n)$

- Time for $\text{Build-Max-Heap}()$:
  - Seems like $O(nh) = O(n \log n)$
  - Actually it is $\Theta(n)$ – (Google for the proof)

- Time for $\text{Extract_max}()$: $O(h)$

- Total sorting time: ______.

- Space? ____.
- Stable? ____.
Performance of Heapsort

- Height $h$ of a heap of $n$ nodes: $h = \Theta(\log n)$

- **Time for** `Build-Max-Heap()`:
  - Seems like $O(nh) = O(n \log n)$
  - Actually it is $\Theta(n)$ – (Google for the proof)

- **Time for** `Extract_max()`: $O(h)$

- Total sorting time: $\Theta(n \log n)$, worst-case.

- Space? $\Theta(1)$, in-place. Stable? No.