Course: Programming II - Abstract Data Types

The ADT Heap

So far we have seen the following sorting types :

1) **Linked List** sort by position	retrieval of an item insertion of an item	- complexity is: O(n) - for random insertion, the complexity is, in the worst case, O(n).
2) **Binary Search Tree** sort by value	retrieval of an item insertion of an item	- complexity O(log n) for each item, O(n*log n) for n items - O(log n) for each item, O(n*log n) for n items
3) **Priority Queues** sort by value	retrieval of an item insertion of an item	- complexity can be constant: O(1) - requires searching algorithms for finding correct position in the Queue

Heaps Slide Number 1

In this lecture we will consider another type of ADT, called Heap, which facilitates efficient insertion and deletion of items to and from a collection of sorted elements. Make sure you don't confuse the Heap ADT with the Heap region of memory used to store objects at run-time!

Let's summarise first the types of sorting we have seen so far. Linked lists are examples of collections of items **sorted by position**. If we consider the most expensive type of insertion, i.e. insertion at an arbitrary position in the list, the worst case scenario would require n access operations to a linked list of n elements. Therefore, the complexity of insertion would be at most of the order O(n), when we need to insert a node at the end of the list. Similarly for the retrieval of an item. Given a position we would require at most n access to the ADT list to retrieve a given item.

A second type of sorting is by value. A binary search tree is an example of an ADT where elements are sorted by (key) value. In this case, the smallest item is the leftmost node in the tree. The complexity of insertion and retrieval is proportional to log(n) when the tree is balanced.

Another type of ADT that we have encountered, where elements are sorted by value, is the priority queue. These are queues where each element has a priority associated with it. Priority Queues can be implemented statically using an array that stores the item in incremental order in the case where the lower value indicates highest priority. So the element to dequeue would be the element with highest priority which is stored at the beginning of the array (i.e. complexity of retrieval is in this case of the order O(1)). Insertion operation would need in this case a binary search to locate the position in the array where to store the new element and still preserve the ordering. Priority queues could also be implemented using a Binary Search Tree in which case the element with highest priority (to dequeue) would be the left-most node in the tree, and the insertion of a node would have the same complexity as the insertion in a Binary Search Tree.

The heap is instead a particular type of ADT that facilitates operations of deletion and insertion by value in a more efficient way than the ways seen so far. We give the definition of a heap in the next slide.

Course: Programming II - Abstract Data Types

An alternative way of sorting

Heaps are complete binary trees that provide an alternative way of sorting a collection of elements.

Different heaps can be defined according to the sorting:

Minheaps: Heaps where the the smallest element is at the root

Maxheaps: Heaps where the the largest element is at the root

How does a heap differ from a binary search tree?

- It is always a complete binary tree
- ordering: a binary search tree can be viewed as a fully sorted set, a heap contains a weaker ordering of the elements.

Heaps Slide Number 2

An ADT Heap is a complete binary tree, with an ordering among its elements that differs from the ordering that we have seen in a binary search tree.

The first important characteristic of a heap is that it is **always a complete binary tree**.

The second characteristic is that the ordering imposed on the elements in the heap is weaker than the ordering required by a binary search tree. In a binary search tree we have seen that all the elements in the left sub-tree have to be smaller than the root element and that all the elements in the right sub-tree have to be bigger than the root element. Considering the in-order traversal on a binary search tree we get the whole set of elements totally ordered.

A **minheap**, instead, is a complete binary tree, such that the smallest element in the given set of elements is at the root of the tree. So the children of the root have to be bigger then or equal to the root elements and both left and right subtrees of the root have to be themselves minheaps.

Another type of heap is the **maxheap**. This is a complete binary tree, where the biggest element of a given collection (according to some method for comparing elements) is at the root of the tree. So the children of the root have to be both smaller then or equal to the root element and both left and right subtrees have to be themselves maxheaps.

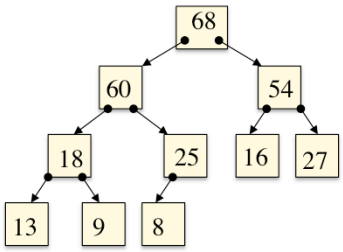
In this lecture we will consider mainly maxheaps. We will therefore use the name heap to mean maxheaps. We will then see the operations of insertion, deletion, and retrieval of an element from a maxheap, which preserves the ordering of the given heap.

Maxheaps are particularly useful to implement **priority queues**.

Note: whereas in a binary search tree there is an ordering requirement between left subtree and right subtree, in an Heap there is only an ordering requirement between a node and its childrens. Moreover, we can allow to have entries with the same key values, whereas in the BST we assumed that key values are unique in a given tree.

Definition

- The **ADT Heap** (or maxheap) is a **complete binary tree** in which the value at the root of the tree is **greater than or equal** to the values of both its children; and both sub-trees are heaps. A heap can also be empty.



How well does this data structure implement a priority queue?

- The element with the largest value is at the root of the tree. This kind of heap is called **maxheap**.
- Retrieving the element with the highest priority would mean returning the root of the tree.

A heap (or **maxheap**) is a complete binary tree:

1a. which is empty, or

1b. whose root contains a key greater than or equal to the keys of its children, and
2b. whose root has heaps as its sub-trees.

In our definition of a heap, the root contains the element with the largest value, i.e. it's a maxheap.

One of the reasons why this particular type of ADT is useful is that it implements priority queues better then a binary search tree can. But how well does this ADT implement a priority queue?

To have an intuitive idea, consider the operation of deleting an element from a priority queue. As mentioned in the first slide, the retrieval operation in a priority queue has to be performed according to the priority value of the items in the queue. Essentially, the element with the highest priority has to be retrieved from the priority queue. If we implement a priority queue using a maxheap, we know that the element with highest priority would be at the root of the tree. So the retrieval operation for the priority queue would just consist of returning the root of the tree. So we would require only one access to the items in the Heap.

Deletion and insertion are more complex than this. To better analyse the efficiency of using heaps for implementing priority queues we need to examine how deletion and insertion operations work for this ADT in more detail. Let's consider first the case of deletion of the element with the highest value. As the (max)heap contains the largest value in the root, the deletion operation for a priority queue could simply be given by removing the root of the heap that implements such a priority queue. Would this be all we have to do? Let's see.

Access Procedures

createHeap()
// post: create an empty heap.

isEmpty()
// post: returns true if the heap is empty, false otherwise.

add(newElem)
// post: adds newElem into a heap in its proper position.
// post: throws exception if heap is full.

removeMax( )
//post: if heap is not empty, returns the element in the root of the heap and deletes it
//post: from the heap. If heap is empty, returns null.

getMax()
// post: retrieves the largest element in the heap, returns null if the heap is empty.

Additional auxiliary procedures to helps us restructure the heap:

heapRebuild(root)
//pre: takes a semi-heap (both subtrees are heaps but root element may not be bigger
//pre: than both its children).
//post: transforms the tree into a heap.

The specific access procedures for an Heap are given above. Note that since a Heap is a complete binary tree, access procedures for binary trees such as `getRootElem()`, `getLeftTree()` and `getRightTree()` could also be used on a heap.

In the lecture on Binary Trees, we saw that complete binary tree can have an efficient static array-based implementation. Since heaps are complete binary trees, the easiest implementation is therefore an array-based implementation. We will look then in the next few slides a static implementation of a Heap.

The auxiliary, and therefore private, procedures such as, for instance, the `heapRebuild` also assumes an array-based implementation. An example of pseudocode for `heapRebuild` is given in slide 10.

Heap Interface for the ADT Heap

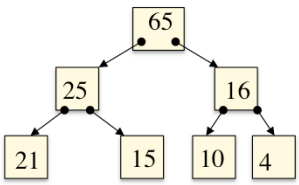
```
public interface Heap<T extends Comparable<T>>{
   public boolean isEmpty();
   //Post: Returns true if the heap is empty, otherwise returns false.
   public void add(T newElem) throws HeapException;
   //Pre: newElem is the new element to be inserted
   //Post: If insertion is successful, newElem is added to the heap at the correct place.
   //post: Throws HeapException if the element cannot be added to the heap.
   public T removeMax( );
   //Post: If heap is not empty, the element with maximum value is returned  and
   //Post: deleted from the heap. If heap is empty, it returns null.
   public T getMax( );
   //Post: If heap is not empty, the element with maximum value is returned.
   //Post: If heap is empty, it returns null.
}
```

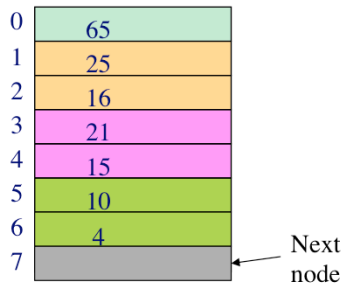

The generic interface `Heap<T extends Comparable<T>>` given in this slide provides the main access procedures for an ADT heap. Note that in this case we have assumed the entry to be an object that implements the interface `Comparable`. We could have used two generic types, one for a comparable key and the other for a generic type associated with the key in a given entry.

In this case the generic declaration would have been similar to what we have given in the case of BST interface. In the case when the key is not an object but a basic type, you will not need to include a generic comparable type in the declaration of the interface, but just the generic type for the value in the entry that is associated with the key.

Implementation of the ADT Heap

Since a heap is a complete binary tree, we can have an efficient static implementation using arrays:





Size of the array = number of elements

The parent of a node(i) is at node (i-1)/2

Children of node(i) are at node(2i+1) and node(2i+2)

As the tree is complete, data is always in consecutive nodes from node 0.

We also need to keep track of the last node (or next available space in the array).

Because a heap is a complete binary tree we can use the array-based implementation of binary trees shown in the lecture notes on Binary Trees, assuming that we know the maximum size of the heap. This slide shows an array-based implementation of a given heap. What we need to know is the size of the array, which is given by the number of the elements in the heap, and the individual elements. These are stored in the array as illustrated in the slide. Main features of this static implementation are:

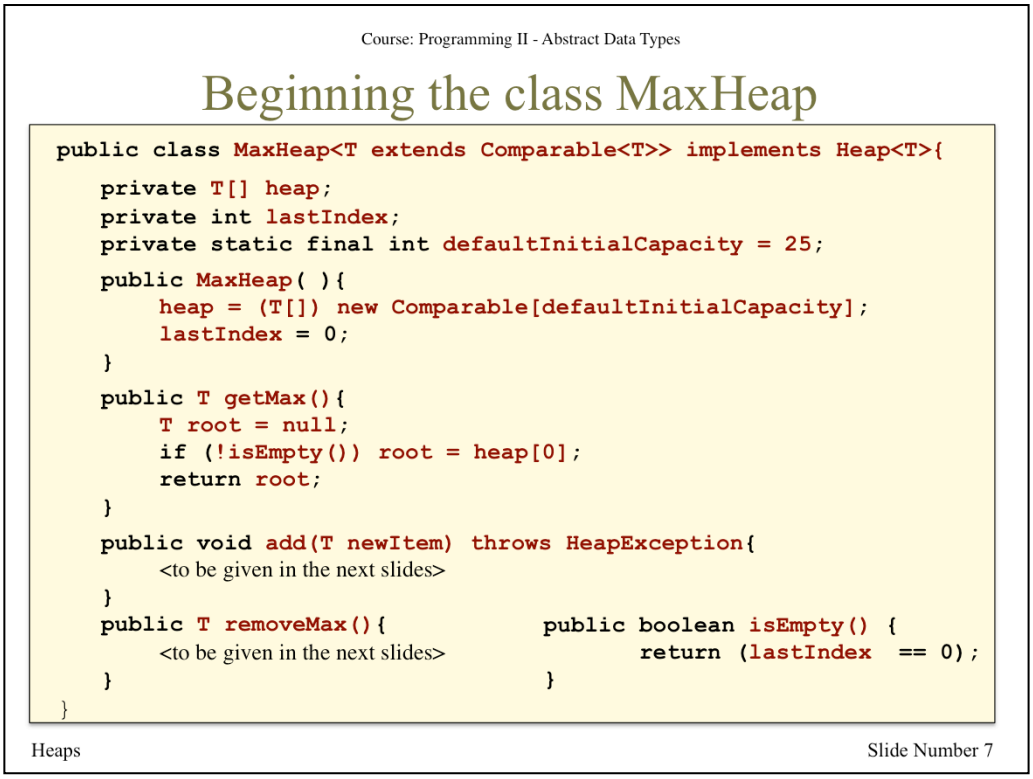
1) Each individual element in the heap is stored in the array **heap**.

2) For each node stored at **heap[i]**, its parent node is located at **heap[(i -1)/2]**

3) The children of the node stored at **heap[i]** are found at **heap[2i+1]** and **heap[2i+2]** elements.

These features allow an efficient implementation of the operations described so far as well as an efficient implementation of the auxiliary procedures for transforming a semi-heap into a heap and for rebuilding a heap after the addition of an new entry, as we will see later on.

However, both the access procedures of `removeMax()` and `add(newElem)` have to guarantee that the resulting ADT is still a Heap. The basic idea behind these two access procedures are illustrated next.

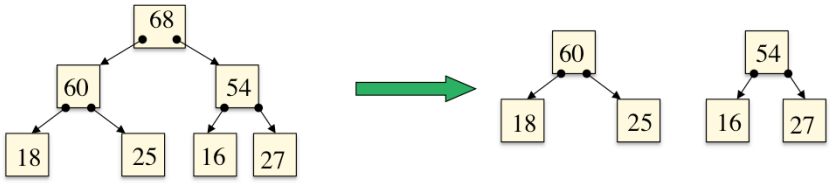


This is the implementation of a class MaxHeap. The basic access procedures are given here. The procedures `add(T newElem)` and `removeMax()`, which are specific to the heap ADTs, are described in the next few slides.

You can already have noticed that the computational time for retrieving an element from a heap is of the order O(1).

Course: Programming II - Abstract Data Types

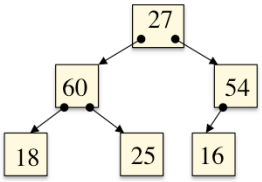
Removing max element: the basic idea



- Retrieving the element with maximum value means retrieving the root of the tree.
- Removing the element with maximum value gives two sub-heaps.

How can we rejoin the two sub-heaps into a single heap?

The simplest way is to take the rightmost node in the tree and put it at the root:



This is a semi-heap: a complete binary tree, whose left and right sub-trees are both heaps, but with the root node out of place in the ordering.

We would therefore need to convert it back to a heap.

Heaps Slide Number 8

The operation of retrieving the element with highest value from a heap is quite simple since it just consists of getting the root of the tree.

The operation of deleting the element with maximum value is slightly more complex because it changes the heap structure and we need to guarantee that after the operation we still have a heap.

When we remove the root of a heap we are left with two disjoint heaps. The delete operation needs therefore to transform the remaining two heaps into a single heap.

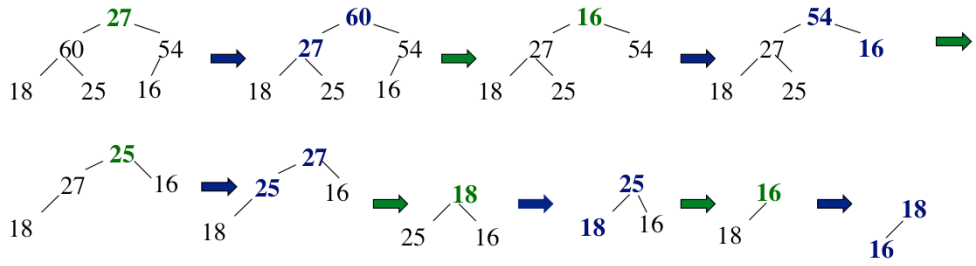
We have to ensure that we have a complete binary tree with the correct ordering property. So, to begin this transformation, we could think of taking the element in the rightmost node of the tree and placing it in the root. This would guarantee that the new heap is a **complete binary tree**. But the result is not necessarily a heap. **It is, however, a complete binary tree whose left and right sub-trees are both heaps**. The only problem is that the element in the root may be (and usually is) out of the heap ordering. Such a structure is called a "**semi-heap**". So we need a way of transforming a semi-heap into a heap. Let's see next slide.

Course: Programming II - Abstract Data Types

Converting a semi-heap into a heap

1. Compare the value in the root with the values of its children,
2. If either child's value is larger than the root, exchange the root with the larger of the two children's values.
3. If there is no exchange, the tree is a heap. Otherwise (i.e. the previous root, now a child, may still be misplaced) repeat steps 1 and 2 until no exchange is necessary, or leaf nodes are reached.

After each element removal we restructure the tree into a valid heap. After the restructuring, the largest value remains at the root, ready for output:



Heaps

Slide Number 9

One strategy for transforming a semi-heap into a heap, is to allow the element in the root to "trickle down" the tree until it finds the first node where its value would be greater than (or equal to) the value of each of its children. To accomplish this, we can first compare the value in the root of the semi-heap with its children's values. If the root value is smaller than the larger of its children's values, then we swap the root value with that of the larger child. In general, more than one such swap is needed in order to rebuild a heap. At each completion of the transformation operation a new heap is generated, the largest value in the binary tree will again be in the root of the newly generated heap, ready for the next delete operation.

A sequence of transformations and delete operations are given in this slide.

The *heapRebuild* auxiliary procedure

```
heapRebuild(root)
//post: Converts a semiheap rooted at index "root" into a heap. Recursively trickles the element at
//post: index "root" down to its proper position by swapping it with its larger child, if the child is larger
//post: than the element. If the element is a leaf, nothing needs to be done.

if (the root is not a leaf) {  // root must have a left child
  child = 2 * root +1          // standard index value for a left child node in static
                               // implementation.

  if (the root has a right child) {
        rightChild = child +1   // index value of right child in the static implementation

    if (heap[rightChild] > heap[child])
          child = rightChild;   // this is the larger child index.
  }

  if (heap[root] < heap[child]) {
        Swap heap[root] and heap[child]
        heapRebuild(child)          //transform semi-heap rooted at child into a heap}
  }

}  // else case: root is a leaf, so we are done.
```


Note: **heap** is the array that stores the elements of the ADT MaxHeap, **root** is the index containing the root node of the (sub)heap, and **size** is the size of the heap.

The delete access procedure

```
T removeMax( ){
   T rootElem = heap[0];
   heap[0] = heap[lastIndex-1];
   lastIndex--;
   heapRebuild(0);
   return rootElem;
}
```

Efficiency:

Removing an element requires us to swap array elements.

How many array elements do we have to swap, at most?

The method `heapRebuild` needs at most $3 * \log_2(n)$ reference changes

So, removeMax( ) is $O(\log_2 n)$, which is quite efficient.

Consider briefly the efficiency of the access procedure. Because the tree is stored in an array, the removal of a node requires you to swap array elements. Since array contents are normally reference variables (in the general cases), swapping will be very fast, because it will just require a change to the value of a reference variable. The data in the objects are not copied individually.

What we need to ask is, at most, how many array elements will we need to swap?

If we consider only the method `heapRebuild`, this method trickles the current root element down the tree until its appropriate place is found. This element travels down a single path from the root to, at worst, a leaf. Therefore, the number of array elements that it might swap is no greater than the height of the tree. Since the tree is a complete tree with n nodes, we know from previous lectures that the height is always $\log_2(n)$. Each swap requires three reference changes, so `heapRebuild` requires at most $3* \log_2(n)$ reference changes. The entire access procedure `removeMax( )` requires then $3* \log_2(n)+1$ reference changes.

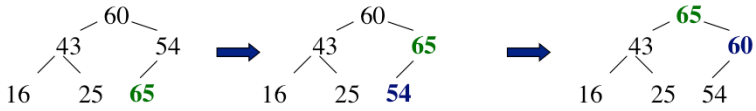
We can conclude that the procedure has a complexity of the order O(log n) which is quite efficient.

Adding an element: the basic idea

The strategy is the opposite of that for delete( ):

1. Insert new element at the bottom of the tree
2. Trickles the new element up to its proper place.

The tree is complete, but 65 is in the wrong place – we must restructure the heap:



The strategy for inserting an element into a heap is the opposite of that for deleting an element. A new element is inserted at the bottom of the tree, and it is then trickled up to its proper place, as illustrated in this slide.

Inserting the element at the bottom of the tree guarantees that the tree is complete after the addition. But the element might be in the wrong place as far as the order with respect to the other entries in the heap. To position the new entry in its correct place it is just sufficient to compare its value with its parent node and the parent of its parent and so on. We can just restrict ourselves to the subtree where we have added the entry, since in a heap there is no relation ordering between the values of the left and right subtrees, but only between chid and parent.

Efficiency of *add*

To trickle an element up the tree is easier. We know that the parent of a node [i] is always at position [(i-1)/2]. So no recursion is needed

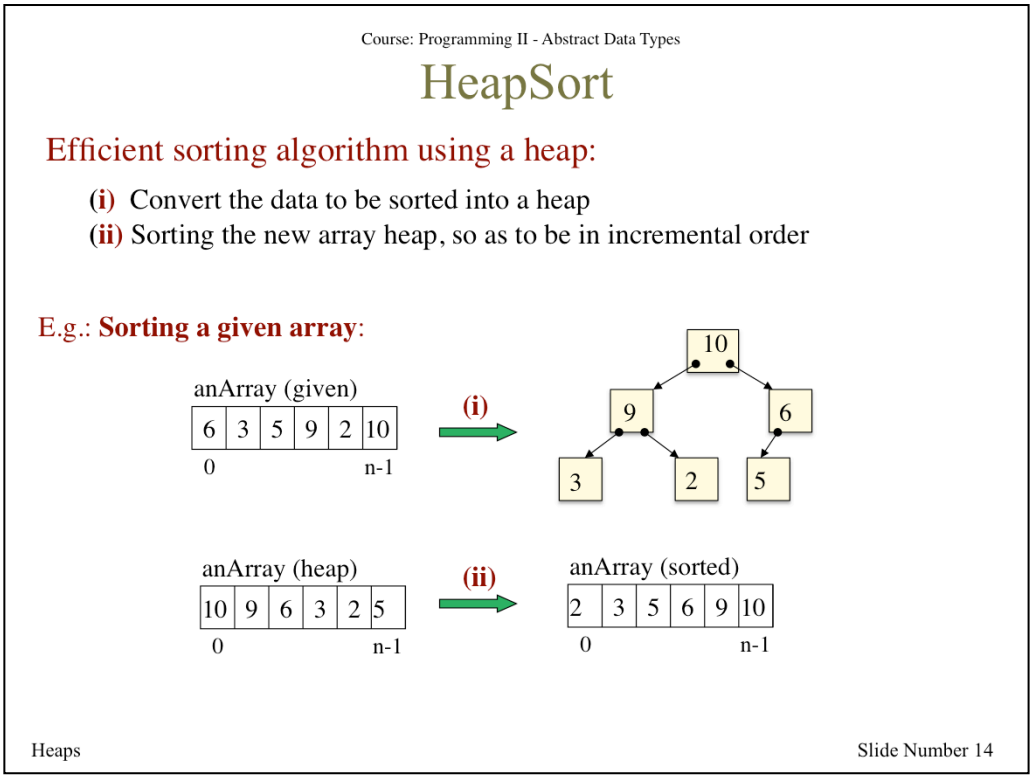
How many array elements do we have to change, at most?

We still trickle the element up along a single path.

We need at most 3 * $\log_2(n+1)$ reference changes.

add(newElem) is also O($\log_2$ n)

The efficiency of this strategy is like that of delete. That is, insert method, in the worst case, has to change array reference elements on a path from a leaf to the root. The number of reference changes, therefore, cannot exceed the height of the tree. Because the height of the tree, which is complete, is always log(n+1), the complexity of the add method is also of the order O(log n).



We have seen so far that an Heap has equal or more efficient operations of insertion, retrieve and deletion than a binary search tree under the assumption that these operations are applied only to the element with highest (or lowest) key value. However, the heap structure does not provide a total ordering over its elements. The `heapSort` is an algorithm that can be used to fully sort the elements in a given `heap`, and more generally for sorting ANY given array. The two important stages of this algorithms are (i) to convert the data into a heap ADT (if it is not already a heap) and (ii) to sort the resulting array heap into a new array with elements in incremental ordering. In this slide we define this algorithm for the particular array-based implementation of the heap. We therefore assume for simplicity that the collection we want to sort is an array of integers given in no particular order.

The first part of the algorithm gives a heap, as shown in (i). This step consists of transforming the array into a heap. One way to do so is to use the "add" access procedure to insert the items into another array-based implementation of a heap one by one. An alternative way could be to apply `heapRebuild` repeatedly on the given array. The fact is that this procedure assumes that the root element of the heap is in the wrong place, but the sub-trees are valid heaps. So starting from the given unordered array, do we have any sub-trees which are heaps? If we look at the leaves of the tree, these are certainly heaps, as they don't have any children. So if we start from the leaves of the complete binary tree representation of our initial array, we could recursively apply the procedure `heapRebuild` on pairs of leaves and their parent and construct heap sub-trees, so proceeding until we reach the root (element with index 0 in the given array). At this stage the array would have elements organised in such as way that the two sub-trees of the root node are heaps and the last recursive application of `heapRebuild` would give the final heap structure.

The code for this alternative way of implementing stage (i) of heapsort is given in the next slide.

After transforming the array into a heap, the **heapSort** algorithm has to sort this array. This is illustrated later.

Converting an array into a heap

pseudocode:

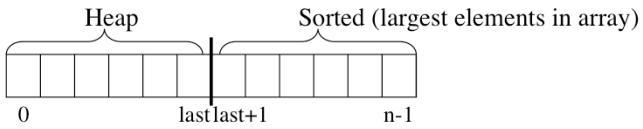
```
Heapsort(anArray, n){
// build initial heap
    for (index = n-1 down to 0)
    //assert: the tree rooted at index is a semiheap

    heapRebuild(anArray, index, n)
    //assert: the tree rooted at index is a heap

<to be continued…>
```

HeapSorting the array Heap (1)

The array is partitioned into two regions: the **heap region** and the **sorted region**.



At the beginning, the sorted region is empty and the heap region is equal to the entire array.

At an arbitrary iteration array[0..last] is the heap region (left still to sort) and array[last+1..n-1] is the sorted region. In particular the sorted region will contain the largest elements of the array.

Invariant:

After step k, the Sorted region contains the k largest values in the initial Heap, and they are in sorted order: Array[n-1] is the largest, ….
The elements in the heap region form a Heap.

HeapSorting the array Heap (2)

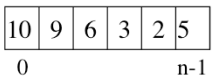
pseudocode:

```
//sort the heap array
last = n-1
//invariant: Array[0..last] is a heap,
             Array[last+1..n-1] is sorted
for (step = 1 through n-1) {
   Swap Array[0] and Array[last]
   Decrement last
   heapRebuild(anArray, 0, last)
}
```

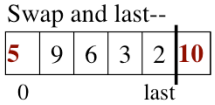

At an arbitrary iteration i of the for loop given here, the largest element in the heap is at the root of the heap. This is at position 0 of the given array. Because of the invariant, all the elements from last+1 up to n-1 are sorted. So to continue the sorting, we need to swap the element in position 0 with the element in position last. This means that now the new sorted area goes from last to n-1, and all the elements in this region of the array are sorted. The new heap area is from 0 to last-1, and because we have changed the first element this new heap area may no longer be a heap. So to fully re-establish the invariant, we need to perform `heapRebuild` on the new heap area.

A trace execution of the sorting loop is given in the next slide.

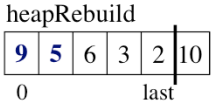
Trace of HeapSort, beginning with:



1) Swap and last--

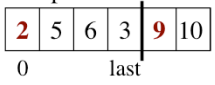


heapRebuild

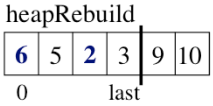


swapping [0] with [5], last = 4, and heapRebuild(array,0,4).

2) Swap and last--

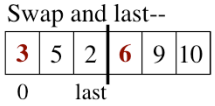


heapRebuild

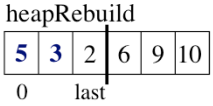


swapping [0] with [4], last = 3, and heapRebuild(array,0,3).

3) Swap and last--

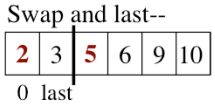


heapRebuild

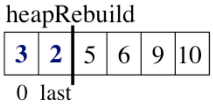


swapping [0] with [3], last = 2, and heapRebuild(array,0,2).

4) Swap and last--

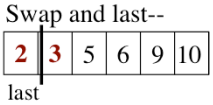


heapRebuild

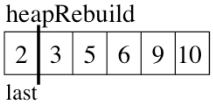


swapping [0] with [2], last = 1, and heapRebuild(array,0,1).

5) Swap and last--



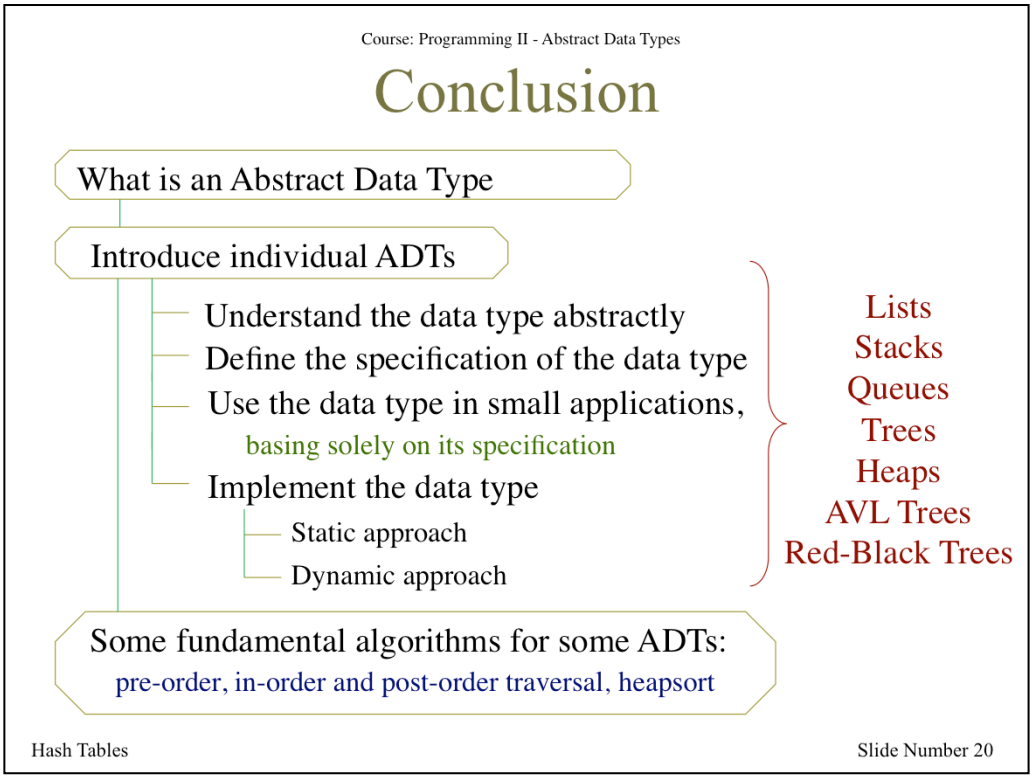
heapRebuild



swapping [0] with [1], last = 0, and heapRebuild(array,0,0).

Summary

- Heaps are particular types of binary trees extremely useful for inserting, and deleting sorted elements. The tree is full except possibly for the lowest level.
- When we know the maximum number of elements stored at any time, array-based implementation is the best heap implementation, useful also to implement priority queues.
- Heapsort is an efficient sorting algorithm, useful for sorting arrays.



We can now conclude this second part of the course with the similar overview slide I gave you in the introduction lecture, but this time as a summary slide of what we have seen and what you are supposed to know. Note that this slide includes AVL Trees and Red-Black Trees as new interesting data structures with clever algorithms. What we haven't covered, and therefore it's not examinable is Hash Tables.


The End