Algorithms & Data Structures

School of Life Sciences, Engineering & Design

Topic:	**Binary Heaps**	Lesson:	**11**
Lecturers:	***Robert de Groote*** ***Dawid Zalewski***	Revision:	**May 17, 2022**

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

Edsger W. Dijkstra

Learning Objectives

After successfully completing this activity a student will be able to:

- Explain what a heap is.
- Explain the differences between a (binary) heap and a (balanced) binary search tree.
- Explain what the main operations performed on a heap are, and their time complexites.
- Understand how a binary heap can be stored in an array.

Team

Before you start this activity, decide on your team members' roles and fill in the table below. Combine the roles of *Spokesperson* and *Reflector* in groups of three.

Team:	**Date**:
Facilitator keeps track of time, assigns tasks and makes sure all the group members are heard and that decisions are agreed upon.	
Spokesperson communicates group's questions and problems to the teacher and talks to other teams; presents the group's findings.	
Reflector observes and assesses the interactions and performance among team members. Provides positive feedback and intervenes with suggestions to improve groups' processes.	
Recorder guides consensus building in the group by recording answers to questions. Collects important information and data.	

Activities

Looking back on the past weeks of this course, *sorted data* is a recurring theme. We have seen how sorting can be used to increase the performance of searching for values in an array, and in the previous two weeks we have seen how this idea can be taken one step further in the form of balanced binary search trees.

In this week, *keeping data sorted* is again our main goal, albeit in a slightly different context. In particular, we'll focus on a situation where we're not interested in quickly finding things in the pool of data, but only need to have fast access to the *greatest* or *smallest* value stored.

The data structure that is the center of attention of this week's activities is the *binary heap*. A binary heap can be seen as a balanced binary search tree, but with much less overhead. This is primarily due to the fact that it does not use any *pointers*, but rather stores all of its values in an *array*.

Handling prioritized tasks

Suppose that you need to keep track of several tasks, each of which has a different *priority*. For example, let's say that we have the following five tasks, along with their priorities:

- Clean the house before your parents come visit. **Priority: 6**
- Buy a gift for your friend's birthday. **Priority: 8**
- Find out more about the binary heap data structure. **Priority: 6**
- Register for next week's exam. **Priority: 10**
- Watch the next episode of your favourite series. **Priority: 1**

If your goal is to finish all these tasks in decreasing order of priority (so finish the activity with the highest priority – register for next week's exam – first), then the way to go through this list is straightforward: work on the task with the highest priority, remove it from the list when done, and repeat.

A to-do list of tasks as given above is, unfortunately, never empty. While working on existing tasks, several new tasks will typically appear - incoming e-mails asking for a document, text messages reminding us of something we've promised, etc. This means that while working on tasks, we also need to adapt our planning based on the incoming tasks - or simply take notice of them and choose not to worry about them for now.

A suitable data structure

What kind of data structure is most suitable for storing our list of prioritized tasks? To answer this question, we first need to think about what kind of operations we would need to perform on this data structure. There are three operations we'd like to perform:

Find-max Find the task with the highest priority. Ideally, this should not take too much effort – if a data structure would force us to iterate over *all* tasks, it would make a poor choice.

Delete-max Remove the task with the highest priority from the data structure (which is what we do after we're finished with the task). This may mean that the data needs to be rearranged to some extent, which could involve some effort.

Insert Insert a new task with a given priority into the structure. Again, we accept that this may involve some rearrangement, but we'd rather keep this to a minimum.

Activity 1: Comparing different data structures

Suppose that you would consider the following three data structures for storing a list of prioritized tasks:

- A sorted array
- A sorted linked list
- A balanced binary search tree

What would be the worst-case time complexities for the three required operations, for each of the three data structures?

Data structure	Find-max	Delete-max	Insert
Sorted array			
Sorted linked list			
Balanced BST			

Question: Which data structure offers the best performance, and why?

If you have completed the previous activity correctly, then you have observed that the two simpler data structures - an array and a linked list - actually perform better on one of the three operations: finding the greatest (or smallest) element. This is due to the fact that in a sorted *linear* structure such as a linked list or array, the minimum and maximum values are stored at the *ends* of the sequence. In a (balanced) BST, however, we must first locate the maximum (or minimum) value, which involves traversing the tree[1].

In the following section, we will see how a new data structure can offer an even better choice than the one you've found to be the most suitable so far, if the three operations are the only ones we need.

Do we need a binary search tree?

In the previous two weeks, we have seen how (balanced) binary search trees (BSTs) provide an excellent way to store sorted data. A balanced BST allows us to find, insert, and remove values with a time complexity of $O(\log n)$, which is great. However, the binary search tree is not the best suited data structure for storing our list of prioritized tasks. This is because the BST keeps *all the values* sorted, but all that we're interested in is the maximum value.

A sorted linked list is another candidate for storing our prioritized task list. We could order the tasks in decreasing order, such that the one with the highest priority comes first. This gives a $O(1)$ time complexity to "find" the maximum.

Because deleting the first node of a linked list is $O(1)$ (it requires only a fixed number of pointer updates, regardless of the number of elements in the list) as well, two of the three operations would be quite efficient.

However, *inserting* a new task with a given priority in the list would require a *linear search* to locate the insertion point of that task. So, for the *insert* operation, the balanced BST is the clear winner.

To conclude, the data structures that we've seen so far have their strong and weak points, but there is not a single one that performs best on *all* three operations. Fortunately, we can combine the benefits of the BST's hierarchical structure with those of having the maximum value stored at the head of a sequence. The data structure that is based on this is called the **binary heap**.

[1]This limitation can be overcome by applying a technique called *memoization*, which involves storing the current maximum value in the tree data structure, and updating it each time that value is removed, or a new (possible greater) value is inserted.

The binary heap: a more relaxed order invariant

Like the binary search tree, the binary heap has hierarchy: each node has up to *two* child nodes. A major difference between a BST and a binary heap, however, is the way in which they *order* their data.

The order invariant of the binary search tree imposes a restriction on which nodes are allowed as child or parent nodes of one another. The binary heap has an order invariant as well, but it is less strict. By relaxing the order invariant of a binary search tree, we obtain the **heap invariant**. The heap invariant states that:

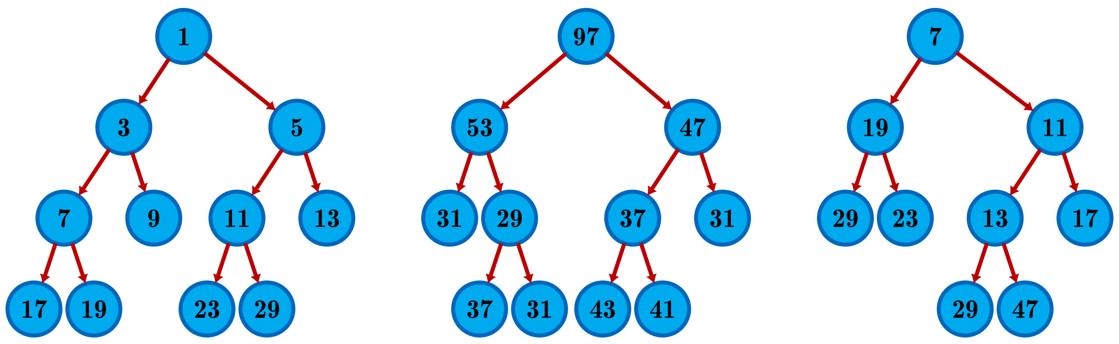
> ***The value stored in any node must be greater than, or equal to, each of its child nodes.***

A binary tree for which this heap invariant holds true is called a (binary) *heap*. A natural consequence of the heap invariant is that the maximum value will be stored in the tree's root node.

The above description applies to one *kind* of heap, called the *max-heap*. If we choose to order elements in the opposite way, meaning that the *minimum* is stored in the tree's root node, then we obtain a *min-heap*. The heap invariant for a min-heap states that the value stored in any node must be *smaller than or equal to* both of its child nodes. In almost all of the activities in this document, we'll be using a *max-heap*, and usually omit the prefix "max-". In the following activity, though, you'll be exposed to both kinds of heaps.

Activity 2: Identify valid heaps

Below, three binary trees are shown[a]. Two of the three are valid heaps.



Which trees are heaps? For each heap, indicate if it's a min-heap or a max-heap. For the tree that is not a heap, describe why it is not a heap.

[a]In the binary trees and heaps that we'll display in this document, we no longer indicate absence of *child* nodes by small red dots, as we've done in the previous weeks. We'll also no longer use a different colour to indicate leaf nodes.

The heap invariant implies an equivalent invariant that must hold for every path from the root node to a leaf node (see Figure 1):

> ***In a heap, the values along every path from the root node to a leaf node are sorted.***

Adding values

The heap invariant gives us more freedom when adding new values to the tree. One of the consequences of this is that we don't need to do any complicated things like rebalancing. In fact, it turns out that, independent of the value that we add, we can always maintain a balanced tree.

To see this, consider the (max-)heap shown below in Figure 2. If we add the value 29 as a right child of the value 5, then the path from the root to 29 contains the values: $19, 13, 5, 29$. In other words, the

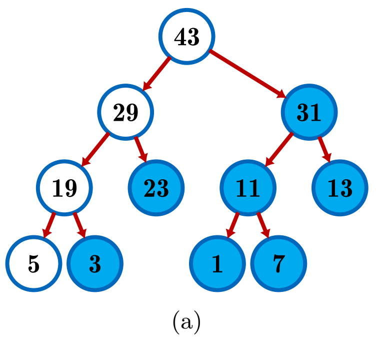
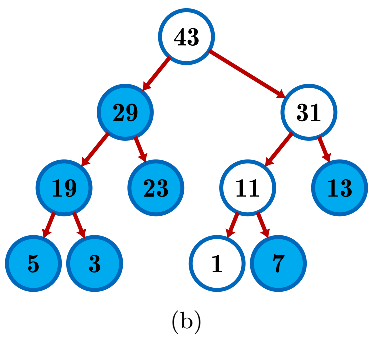
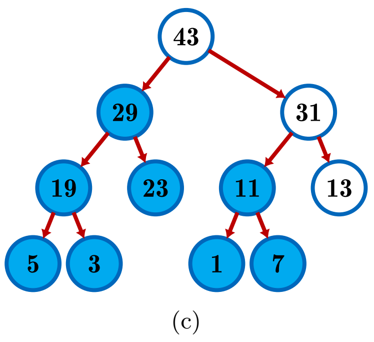
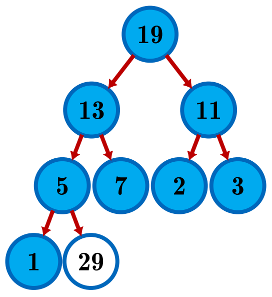
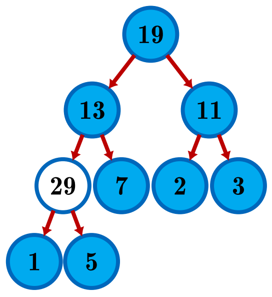




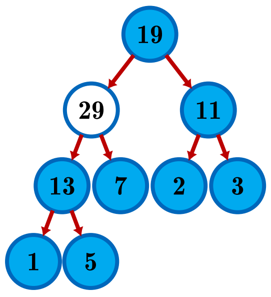
Figure 1: In a binary (max-)heap, every path from the root node to a leaf node is sorted in descending order. Here, three of the six paths (white nodes) in an example heap are shown.

path is no longer sorted: 29 is the greatest of the four values, and must thus be the first value on the path, not the last.

To fix this, we can iteratively *swap* the violating value with its parent value, until the path forms a sorted list of values again. This is guaranteed to restore the heap invariant for the entire tree.


(a) Added 29.


(b) Swapped 29 and 5.


(c) Swapped 29 and 13.

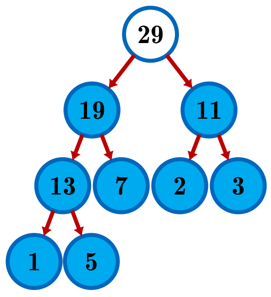

(d) Swapped 29 and 19.

Figure 2: New values are added to the binary heap as child nodes. The heap invariant is then restored by bubbling the new value upwards. In the worst case, the new value needs to bubble up al the way to the root node.

Activity 3: Do it yourself

In the max-heap of Figure 2d, suppose that the value 12 is added as a right child of 3. How many, and which specific swaps are needed to restore the heap invariant?

Now that we've covered how to add new values (i.e., the *insert* operation) to a binary heap, we can focus on the next operation: *delete-max*.

Removing the maximum value

In a binary heap, the only value that we ever plan te remove from the tree is the value in the *root node*. In case the heap is a min-heap, then this value is the smallest value in the tree; in case of a max-heap it is the greatest value.

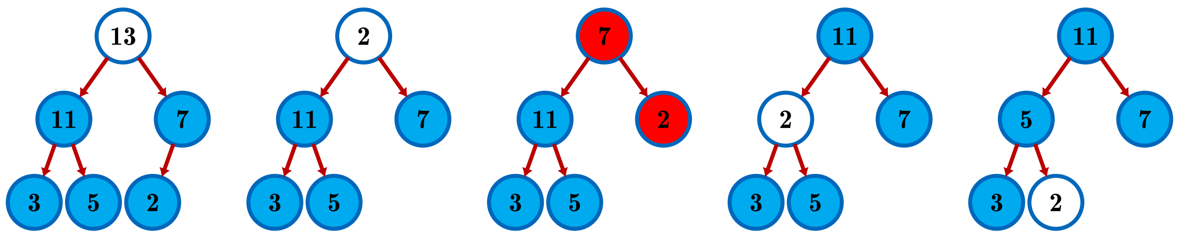
For a binary search tree, removing the value stored in the root node of the tree (assuming that the root node has two child nodes) is done by replacing the value with another value, such that the order invariant is kept intact.

For a heap, the procedure is somewhat different: we replace the root node's value with the value stored in *any* of the *leaf* nodes (and then delete that leaf node). Because this may cause a violation of the heap invariant, we apply a correction similar to the one we applied after inserting a new value.

One main difference with the previous operation (adding a value), is that values are now swapped *downwards*, i.e., towards the leaf nodes of the tree. This means that there are (at most) *two* swapping candidates: the *left* and the *right* child. In order to guarantee that the heap invariant is restored, we must swap with the ***maximum*** of the two child values. If we do not do this (and choose the maximum value instead), then the swap creates a new problem.

This is illustrated in Figure 3 shown below, where the maximum value, 13, is removed from the heap, replacing it with the value of 2. The value 2 is smaller than both of its children – 11 and 7, so we must decide which value to swap with. If we would swap 2 with 7 (i.e., the *smallest* child), then we would immediately introduce a violation, because 7 would then become the parent of 11. Since 7 is smaller than 11, this would thus violate the heap invariant.

We therefore swap 2 with its greatest child value – 11, and subsequently with 5. The heap invariant is then restored again, with the value 11 as the new maximum.



(a) About to remove the max, 13. (b) Moved 2 into the root. (c) Swapped 2 and 7 (incorrect). (d) Swapped 2 and 11 (correct). (e) Swapped 2 and 5.

Figure 3: When removing the maximum (*delete-max*) from a heap, the root value is replaced by one of the leaf values. The heap invariant is then restored by bubbling the new root value downwards. If a value is smaller than any of its child values, then the value is swapped with the *maximum* child value.

Activity 4: Worst-case time complexities

What are the worst-case time complexities of:

- Finding the maximum value
- Removing the maximum value
- Inserting a new value
- Finding an arbitrary value

for a binary max-heap and for a balanced binary search tree? Give your answer in Big O notation.

Data structure	**Find-max**	**Delete-max**	**Insert**	**Find**
Balanced BST				
Binary max-heap				

Binary heaps and arrays

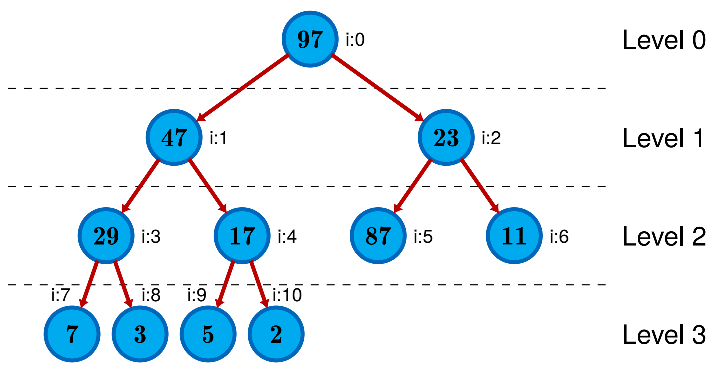
In the previous section, we have seen that a lot of the complexity involved with a self-balancing binary search tree is no longer necessary for a binary heap. The heap invariant gives us more freedom in choosing where to insert new values: we can add the new value as a new leaf node, and then restore the heap invariant by a few *upward* swaps. Also, when removing the value stored in the root, we can choose any of the leaf node values as a replacement, and restore the invariant through *downward* swaps.

Consequently, maintaining *balance* - which is required for efficient repair of any violation of the heap invariant - is much less of an issue.

Because complex modifications of the structure of the tree such as removal of non-leaf nodes and rotations are no longer necessary in a binary heap, there's also no longer any reason for using pointers to link nodes to parent and child nodes. In fact, we can simplify our binary heap data structure by storing its contents in an *array*. Apart from making things simpler, this even has some more benefits in terms of overhead, because an array takes up a contiguous block in memory, improving locality of reference, leading to fewer cache misses, etc.

From nodes to indices

In order to store a binary heap in an array, each node needs to be assigned an *index* into the array. A straightforward approach to do this is *level by level*, storing each level of nodes (starting with the level containing the root node) in left-to-right order. This means that the root node's value is stored at index 0, the values of its left and right child at indices 1 and 2, the four nodes in the next level at indices 3 to 6, etc. Figure 4 illustrates this idea for a binary heap consisting of 11 nodes.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
97	47	23	29	17	87	11	7	3	5	2					

Figure 4: The values of a binary (max-)heap can be stored in an array, by going through the heap from top to bottom, and left to right. The heap invariant allows that each level of the heap can be filled from left to right, and one level can be fully filled before starting a next.

Implementation time

The source code provided for this week's activities consists of multiple classes. The class `maxheap`↩ contains an implementation of a vector-backed (the C++ `vector` is a dynamic array) max-heap. A partial definition (for the complete definition, see the `maxheap.h` header file) is given below. A more detailed description of each of these functions will be given in the upcoming activities.

```
class maxheap {
public:
  const task& maximum() const; // returns the task with the highest priority
  void delete_max();     // deletes the task with the highest priority
  void insert(const task& value); // inserts a new task
private:
  size_t heapify();
  size_t left_child_index(size_t index) const;
  size_t right_child_index(size_t index) const;
  size_t parent_index(int index) const;
  size_t bubble_up(int index);
  size_t bubble_down(int index);
  void swap(size_t first_index, size_t second_index);
};
```

Values stored in the heap are not numbers, but they are `task`s. The `struct task` is a very simple class to represent a prioritized task:

```
struct task {
  int priority;
  std::string description;
};
```

The priority is an indication of the urgency of finishing this task, compared to other tasks. The higher the priority, the more urgent the task is. This means that, in our `max_heap`, the task that needs to be finished first will be stored in the root node of the heap.

Locating children and parents

When using an array to store the contents of a binary heap, then level 0 (i.e., the root node) contains index 0; level 1 consists of indices 1 and 2, and so on. In general, level k consists of indices $2^k - 1$ up to, but excluding, $2^{k+1} - 1$.

The contents of a binary heap are stored contiguously in the array. In other words, we do not leave any "gaps" in the array. This means that when adding new values to the heap, these will be added to the first available index, which corresponds to the leftmost "free" node at the deepest, currently unfilled, level.

This level-by-level assignment of indices to nodes allows us to compute the index of any given node's parent or (left or right) child, by making a simple calculation.

Activity 5: Index computations

The `private` functions `left_child_index` and `right_child_index` of the `maxheap` class return the index of the left or right child of any node given by its index. Similarly, the function `parent_index` returns the index of a node's parent.
These functions are not yet implemented - implement them so that, for a node with a given index,

- `left_child_index` returns the index of the node's left child,
- `right_child_index` returns the index of the node's right child,
- `parent_index` returns the index of the node's parent.
- The function return the constant `maxheap::npos` if the node does not have the requested child / parent.

Use the code listed below to test your implementation.

```
int main() {
  heap_tester::test_left_child_index();
  heap_tester::test_right_child_index();
  heap_tester::test_parent_index();
}
```

The maximum value of a binary (max-)heap is stored in the root node. This means that the *find-max* operation, which gives the maximum value of the heap, consists of a single lookup, with (constant) time complexity $O(1)$. The find-max operation is not too difficult to implement, but it is not yet in the `maxheap` class.

Activity 6: Implement the find-max operation

The *find-max* operation of our binary heap is represented by the `maximum` member function. Implement the `maximum` function of the `maxheap` class.

Use the code listed below to test your implementation.

```
int main() {
    heap_tester::test_find_max();
}
```

Fixing violations of the heap invariant

New values are inserted into the heap in a level-by-level, and left-to-right order. In our implementation, the function `insert` of the `maxheap` class adds a new value to the heap, by *appending* it to the array.

This may cause a violation of the heap invariant in case the added value is greater than its parent value. To repair the heap, the path from the root node to the newly inserted leaf node must be brought back into sorted order. This can be done by moving the value upwards in the tree, swapping it with its parent value at each step, as shown earlier in Figure 2.

Activity 7: Bubbling up

Our binary heap's `insert` function is already given. It appends the new value to the array, and then repairs the binary heap by "bubbling up".

```
size_t maxheap::insert(const task& value) {
  m_values.push_back(value);
  bubble_up(m_values.size() - 1);
}
```

Implement the `bubble_up` function, which must repair any heap violation caused by insertion of the value: if the value is greater than its parent value, then these two values are swapped. This is repeated until the heap invariant is restored again.

The function must return the number of *swaps* that were performed. To swap two values, you can use the `private swap(size_t, size_t)` function of the `maxheap` class, which swaps the two tasks stored at the given indices.

Use the code listed below to test your implementation.

```
int main() {
  heap_tester::test_bubble_up();
}
```

In case the maximum value is deleted from the heap, its value is replaced by the value in the rightmost leaf node in the deepest level of the heap. The rightmost leaf node is then deleted from the tree, by removing the last element from the array.

Replacing the maximum value with another value will most probably cause a violation of the heap invariant - there's a good chance that a value from the deepest level of the tree is smaller than the value stored in the root. Fixing this violation is only slightly more complicated than the steps we've taken when inserting a new value. This was discussed earlier, and was illustrated in Figure 3.

Activity 8: Bubbling down

The `maxheap` class already has a function `delete_max`, which deletes the maximum value from the heap by swapping it with the last value in the array, and removing the array's last element. The function then calls the `bubble_down` function to repair any violation of the heap invariant this may have caused:

```cpp
void maxheap::delete_max() {
  std::swap(m_values[0], m_values.back());
  m_values.pop_back();
  bubble_down(0);
}
```

Implement the `bubble_down` function. This function must move the new root value downwards, by swapping it with the minimum of its child values (but only if it is greater than any of its child values!).

Like the `bubble_up` function, it must return the total number of swaps that were performed. Use the `swap(size_t, size_t)` function, and use the code listed below to test your implementation.

```cpp
int main() {
  heap_tester::test_bubble_down();
}
```

Building a heap

Imagine that you come back from a well-deserved long holiday, only to find your inbox full of tasks that demand your attention. You'll need to sit down for a few minutes to determine the order in which all these tasks need to be completed. To build a heap from the tasks, the logical option may seem to start with an *empty* binary max-heap, and insert each of the tasks until they are all in the heap. Next, you can go through the tasks by iteratively finding and deleting the highest-priority task.

Building a heap from an unsorted sequence of n values in the way described above, basically performs n insertions into a heap. Each of these insertions has a worst-time time complexity of $O(\log m)$, with m being the number of elements that are in the heap – note that m will increase after each insertion. The total time complexity to build a heap in this way is thus $O(\log 1 + \log 2 + \ldots + \log(n-1))$, which is the same as $O(\log{(n-1)!})$[2].

As an alternative, rather than inserting values one by one into the binary heap, we may just as well copy all the values into the array that stores the heap's values (i.e., without taking care that the heap invariant is maintained), and fix all heap violations in a subsequent pass. As it turns out, this second approach is a much better way – in terms of performance – to build a heap from unsorted values.

The worst-case number of heap violations in a *max*-heap is triggered when we copy into the heap's internal storage a sequence of values that is sorted from low to high (note that this the *best-case* situation for a *min*-heap).

This is illustrated in Figure 5: each of the values violates the heap invariant. Consequently, multiple swaps need to be performed in order to restore the invariant.

Heapify

The process of building a heap from a sequence of values is referred to as the *heapify* operation. It involves bubbling down all nodes, starting with the leaf nodes, ending with the root node. Depending

[2]The logarithm of a factorial has been well-studied in mathematics - see Stirling's approximation.

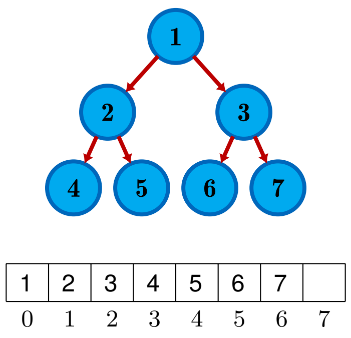


Figure 5: When initializing a binary *max*-heap from a list of values that are ordered in ascending order, there are many heap violations (each child nodes is now greater than its parent, which violates the heap invariant) that need to be repaired.

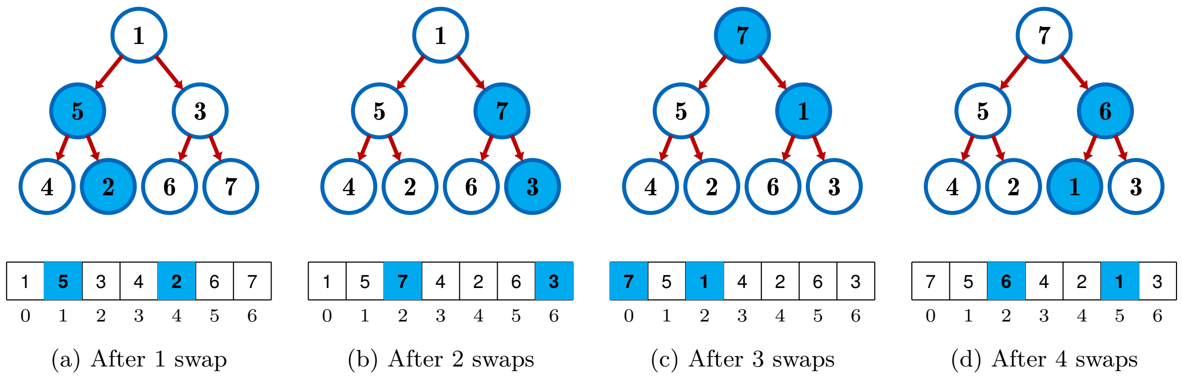


Figure 6: Restoring the heap invariant for the heap of Figure 5, in a bottom-up approach – heap violations of child nodes are fixed before parent nodes. This requires a total of four swaps.

on the values stored in the nodes, some values do not need to be swapped at all, whereas for others multiple swaps need to be applied to put them at the right position.

This is illustrated in Figure 6, which shows how the tree of Figure 5 is changed into a heap, by restoring the heap invariant through six swaps.

The number of swaps needed to restore the heap invariant, after filling the heap with many values, greatly depends on the order in which violations are fixed. In Figure 6, the optimal strategy is used.

Activity 9: Implement heapify

The previous paragraphs and figure explain and illustrate the *heapify* operation, which creates a heap from an unordered sequence. The operation involves visiting child nodes before their parent nodes, bubbling values down when necessary.

The heapify operation is represented by a (private) member function (which is, unsurprisingly, named `heapify`) of the `maxheap` class. The function is called by the `maxheap` *constructor*:

```
maxheap::maxheap(const std::vector<task> &values) : m_values{values} {
  heapify();
}
```

Implement the function `maxheap::heapify`. The function must return the total number of swaps performed. Use the code listed below to test your implementation:

↪ *Activity continues on the next page.*

```
int main() {
  heap_tester::test_heapfiy();
}
```

The heapify operation iterates all nodes in the tree, calling the `bubble_down` function for each index. The latter, on its turn, performs several swaps. Consequently, we can "measure" the worst-case time complexity of the heapify operation by counting the number of swaps performed, if we're able to set up a scenario where the number of required swaps is the highest.

Activity 10: Heapify - time complexity

The worst-case scenario for the heapify operation is when it needs to build a (max-)heap from values that are in *increasing* order.

Carry out the following experiment: write a short program that generates a `std::vector` filled with tasks (see the `heap_tester::test_heapify` function), ordered by increasing priority. Next, add some code to the `maxheap` constructor so that it prints the number of swaps performed by the `heapify` operation.

Vary the length of the generated vector, and record the number of swaps performed during the heapify operation into the table below.

Number of values	Number of swaps
5	3
10	
20	
50	
100	
200	
300	
400	
500	
1000	

Question: what is the worst-case time complexity of the heapify operation? Use the completed table to motivate your answer.

Heaps and sorting

Up to this point, we have not really done much with the tasks that are stored in our heap. The primary use case is of course that we'd like to go through all of them, in decreasing order of priority. This is in fact quite easy, and can be achieved by the following code:

```
maxheap heap = { /* tasks */ };
while (heap) {
  std::cout << "Next task: " << heap.maximum().description << std::endl;
  heap.delete_max();
}
```

What this short fragment of code basically does is *sorting* all the tasks in order of decreasing priority, using a heap. Deleting the maximum value *swaps* puts the current greatest value at the current "end" of the array or vector – it will stay at that location if we do not insert any new values into the heap.

Successively deleting the maximum value from a heap puts increasingly smaller values at decreasing indices of the array or vector, until the heap is empty – leaving the smallest value at index 0. In other words, successive deletions of the maximum effectively *sorts* the contents of an array!

Of course, there's no real need to first *copy* the values from a vector into an instance of our `maxheap` - we can just use the vector (in our example C++ code) itself as the heap's storage. In other words, the `heapify` and `bubble_down` functions can be run on any array or vector, by *interpreting* it as a heap. You'll reimplement `heapify` and `bubble_down` in the following two activities – if you've successfully completed these implementations as member functions, then should primarily be a matter of copy-pasting.

Activity 11: Bubbling down vector elements

In your `main.cpp` file, implement the following two functions:

bubble_down – which takes a `vector` of `int`s, an `index`, and a `count` parameter. The function must treat the first `count` elements of the `vector` as a *max-heap*, and restore heap violations from index `index` by swapping values downwards. The function does not return a result.

heapify – which takes a `vector` of `int`s. The function must rearrange the contents of a vector so that it represents a max-heap. The function should make use of the `bubble_down` function, and does not return a result.

For these two functions you can reuse most of the code in the corresponding member functions of the `maxheap` class, but some minor changes are required.

Use the code listed below to test your functions:

```cpp
void bubble_down(std::vector<int> &values, size_t index, size_t count);
void heapify(std::vector<int> &numbers);

int main() {
  std::vector<int> heap = {9, 7, 6, 3, 2, 5};
  std::swap(heap[5], heap[0]);
  bubble_down(heap, 0, 5);
  assert(heap == (std::vector<int>{7, 5, 6, 3, 2, 9}));
  std::swap(heap[4], heap[0]);
  bubble_down(heap, 0, 4);
  assert(heap == (std::vector<int>{6, 5, 2, 3, 7, 9}));
  std::swap(heap[3], heap[0]);
  bubble_down(heap, 0, 3);
  assert(heap == (std::vector<int>{5, 3, 2, 6, 7, 9}));
  std::swap(heap[2], heap[0]);
  bubble_down(heap, 0, 2);
  assert(heap == (std::vector<int>{3, 2, 5, 6, 7, 9}));
  std::swap(heap[1], heap[0]);
  assert(heap == (std::vector<int>{2, 3, 5, 6, 7, 9}));

  heap = {2, 9, 5, 3, 7, 6};
  heapify(heap);
  assert(heap == (std::vector<int>{9, 7, 6, 3, 2, 5}));
}
```

Now that we have created *heapify* and *bubble up* operations that are not coupled to the `maxheap` class, we can use these functions to sort a `vector` (i.e., dynamic array) of integer numbers.

Activity 12: In-place heap sort

Suppose that a max-heap of n elements is stored in an array. To delete the maximum value from this max-heap, we swap the value at index 0 with the value at index $n-1$, and then restore the heap invariant by bubbling up the value at index 0, in a *smaller* heap.

If we again were to delete the maximum value, it would be stored at index $n-2$, the next highest would be placed at index $n-3$ and so on. In other words, by using the `swap` and `bubble_up` functions iteratively, it is possible to sort an array.

This approach to sorting is called *heap sort*, and has a (worst-case) time complexity of $O(n \log n)$. Heap sort involves performing the heapify operation, followed by iterative deletion of the maximum element.

Write a function (e.g., in the `main.cpp` file) named `heap_sort` that takes a `vector<int>` as its sole parameter. The function must sort the vector in ascending order (lower values precede higher values), by combining the `heapify` and `bubble_down` functions.

Use the code listed below to test your implementation.

```cpp
void heap_sort(std::vector<int>& values);

int main() {
  std::vector<int> numbers{2, 9, 5, 3, 7, 6, 0, 4, 1};
  heap_sort(numbers);
  assert(numbers == (std::vector<int>{0, 1, 2, 3, 4, 5, 6, 7, 9}));
}
```

Generalizing things

Being able to sort a `vector` is nice, but when it comes to wide applicability, it doesn't score too many points. What if we'd like to sort a `vector` of `string`s instead? Or a sequence of `task`s? With the current code, we'd have to go through the tedious process of copy-pasting the code that we have for `int`s, and substitute all references to the `int` type to whatever type we're interested in. There are some limitations, of course – the type of the elements in our `vector` must be *comparable*, i.e., we must be able to use the greater-than (>) operator to see which value should come first.

Fortunately, we can use an important C++ language feature that allows us to avoid such a frustrating and repetitive job: *templates*. Templates are functions or classes that have *parameters* which can be substituted with a *type* (such as `int`, `std::string`, or `task`). Without going too much into detail, you'll use this language feature to generalize your `heap_sort` function.

Activity 13: General heap sort

Make your code generic by turning the `heap_sort`, `heapify` and `bubble_down` functions into *templates*. To do this, replace the type `int` with a `T` (or some other identifier), and write `template<typename T>` in the line directly above the function. This indicates that the function's code is a template that will be generated once it used with a specific type, which will substitute the `T` parameter.

↪ *Activity continues on the next page.*

If necessary, change all the `int` type specifiers within the templated functions with a `T`.

To test your templates, use the following code to sort `vector`s of various types:

```cpp
template<typename T>
size_t bubble_down(std::vector<T> &values, size_t index, size_t count);

template<typename T>
size_t heapify(std::vector<T> &numbers);

template<typename T>
void heap_sort(std::vector<T> &numbers);

int main() {
  std::vector<int> numbers{2, 9, 5, 3, 7, 6, 0, 4, 1};
  heap_sort(numbers);
  for (size_t i = 1; i < numbers.size(); ++i)
    assert(!(numbers[i - 1] > numbers[i]));

  std::vector<char> letters{'z', 't', 'x', 'y', 'a', 'f', 'q', 'c', 's'};
  heap_sort(letters);
  for (size_t i = 1; i < letters.size(); ++i)
    assert(!(letters[i - 1] > letters[i]));

  std::vector<std::string> strings{"foo", "bar", "bob", "ham", "bacon"};
  heap_sort(strings);
  for (size_t i = 1; i < strings.size(); ++i)
    assert(!(strings[i - 1] > strings[i]));
}
```