WS 2007/2008

Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

1. Heapsort

Definition 1

A heap is an almost complete binary tree whose vertices are annotated with key values such that the heap condition is satisfied in each vertex v: The key value stored in v is at most as large as the key values stored in v's children.
Hence, the root of the heap is annotated with a minimum key value. And each path of vertices from the root to a leaf is annotated with increasing sequence of keys.

A data structure is a structued method of storing data elements (typically permitting efficient access to its contents), along with a set of operations that allow access to the data structure and manipulation of the structure in such a way that the storage organization remains intact. We shall now see how to define a set of operations on heaps which will help us write down the HeapSort algrithm in just 4 lines of code.

1. Heapsort

Definition 1

A heap is an almost complete binary tree whose vertices are annotated with key values such that the heap condition is satisfied in each vertex v: The key value stored in v is at most as large as the key values stored in v's children.
Hence, the root of the heap is annotated with a minimum key value. And each path of vertices from the root to a leaf is annotated with increasing sequence of keys.

A data structure is a structued method of storing data elements (typically permitting efficient access to its contents), along with a set of operations that allow access to the data structure and manipulation of the structure in such a way that the storage organization remains intact. We shall now see how to define a set of operations on heaps which will help us write down the HeapSort algrithm in just 4 lines of code.

1.1 Operations on Heaps

The three heap operations that we need for sorting keys are

1. void reheap (heap h) : repair a "heap" in which the heap condition is violated at the root
2. heap create_heap (key $A[]$, unsigned n) : construct a heap from an array of key values
3. key delete_min(heap h) : delete the key stored at the root and restore the heap

1.1 Operations on Heaps

The three heap operations that we need for sorting keys are

1. void reheap (heap h) : repair a "heap" in which the heap condition is violated at the root
2. heap create_heap (key $A[]$, unsigned n) : construct a heap from an array of key values
3. key delete_min(heap h) : delete the key stored at the root and restore the heap

1.1 Operations on Heaps

The three heap operations that we need for sorting keys are

1. void reheap (heap h) : repair a "heap" in which the heap condition is violated at the root
2. heap create_heap (key $A[]$, unsigned n) : construct a heap from an array of key values
3. key delete_min(heap h) : delete the key stored at the root and restore the heap

1.1.1 The reheap-Operation

- Suppose an almost complete undirected binary tree with vertex annotations is given which satisfies the heap condition at every vertex except the root.
- Let v be the tree's root.
- Hence, the key stored at v is not $\leq$ the keys stored at both of v's children (if they exist).
- Let v^* be the child of v that has the smaller key.
- Strategy: We exchange the keys of v and v^*. Then the same procedure is applied recursively to the subtree below v.

1.1.1 The reheap-Operation

- Suppose an almost complete undirected binary tree with vertex annotations is given which satisfies the heap condition at every vertex except the root.
- Let v be the tree's root.
- Hence, the key stored at v is not $\leq$ the keys stored at both of v's children (if they exist).
- Let v^* be the child of v that has the smaller key.
- Strategy: We exchange the keys of v and v^*. Then the same procedure is applied recursively to the subtree below v.

1.1.1 The reheap-Operation

- Suppose an almost complete undirected binary tree with vertex annotations is given which satisfies the heap condition at every vertex except the root.
- Let v be the tree's root.
- Hence, the key stored at v is not $\leq$ the keys stored at both of v's children (if they exist).
- Let v^* be the child of v that has the smaller key.
- Strategy: We exchange the keys of v and v^*. Then the same procedure is applied recursively to the subtree below v.

1.1.1 The reheap-Operation

- Suppose an almost complete undirected binary tree with vertex annotations is given which satisfies the heap condition at every vertex except the root.
- Let v be the tree's root.
- Hence, the key stored at v is not $\leq$ the keys stored at both of v's children (if they exist).
- Let v^* be the child of v that has the smaller key.
- Strategy: We exchange the keys of v and v^*. Then the same procedure is applied recursively to the subtree below v.

1.1.1 The reheap-Operation

- Suppose an almost complete undirected binary tree with vertex annotations is given which satisfies the heap condition at every vertex except the root.
- Let v be the tree's root.
- Hence, the key stored at v is not $\leq$ the keys stored at both of v's children (if they exist).
- Let v^* be the child of v that has the smaller key.
- Strategy: We exchange the keys of v and v^*. Then the same procedure is applied recursively to the subtree below v.

Algorithm:

void reheap(heap h){
  v :=root of h
  while (heap-condition not satisfied at v) do
    v^* :=child of v with the smallest key
    exchange keys of v and v^*
    $v := v^*$
  od
}

- Once a leaf has been reached the heap condition is trivially satisfied. Therefore the procedure terminates.
- Correctness follows from the fact that after each iteration, the subtree in which the heap condition is violated strictly shrinks.
- Complexity: Each iteration costs constant time. The complexity is therefore proportional to the height/depth of the tree. If h has n vertices then the time complexity is $O(\log n)$.

Algorithm:

void reheap(heap h){
  v :=root of h
  while (heap-condition not satisfied at v) do
    v^* :=child of v with the smallest key
    exchange keys of v and v^*
    $v := v^*$
  od
}

- Once a leaf has been reached the heap condition is trivially satisfied. Therefore the procedure terminates.
- Correctness follows from the fact that after each iteration, the subtree in which the heap condition is violated strictly shrinks.
- Complexity: Each iteration costs constant time. The complexity is therefore proportional to the height/depth of the tree. If h has n vertices then the time complexity is $O(\log n)$.

Algorithm:

void reheap(heap h){
  v :=root of h
  while (heap-condition not satisfied at v) do
    v^* :=child of v with the smallest key
    exchange keys of v and v^*
    $v := v^*$
  od
}

- Once a leaf has been reached the heap condition is trivially satisfied. Therefore the procedure terminates.
- Correctness follows from the fact that after each iteration, the subtree in which the heap condition is violated strictly shrinks.
- Complexity: Each iteration costs constant time. The complexity is therefore proportional to the height/depth of the tree. If h has n vertices then the time complexity is $O(\log n)$.

1.1.2 The delete_min-Operation

- We need to delete a key of minimum value, stored at the root
- We replace the root by the only vertex that can be removed without invalidating the graph's property of being an almost complete binary tree: the rightmost vertex on the deepest level
- This violates the heap condition (only) at the root. reheap()
- Complexity: $O(d(h)) = O(\log n)$

Algorithm:

key delete_min(heap h){
  v :=root of h
  $k := v.key$
  v' :=rightmost leaf on h's deepest level
  $v.key := v'.key$
  delete v'
  reheap(h)
  return k
}

1.1.2 The delete_min-Operation

- We need to delete a key of minimum value, stored at the root
- We replace the root by the only vertex that can be removed without invalidating the graph's property of being an almost complete binary tree: the rightmost vertex on the deepest level
- This violates the heap condition (only) at the root. reheap()
- Complexity: $O(d(h)) = O(\log n)$

Algorithm:

key delete_min(heap h){
  v :=root of h
  $k := v.key$
  v' :=rightmost leaf on h's deepest level
  $v.key := v'.key$
  delete v'
  reheap(h)
  return k
}

1.1.2 The delete_min-Operation

- We need to delete a key of minimum value, stored at the root
- We replace the root by the only vertex that can be removed without invalidating the graph's property of being an almost complete binary tree: the rightmost vertex on the deepest level
- This violates the heap condition (only) at the root. reheap()
- Complexity: $O(d(h)) = O(\log n)$

Algorithm:

key delete_min(heap h){
  v :=root of h
  $k := v.key$
  v' :=rightmost leaf on h's deepest level
  $v.key := v'.key$
  delete v'
  reheap(h)
  return k
}

1.1.2 The delete_min-Operation

- We need to delete a key of minimum value, stored at the root
- We replace the root by the only vertex that can be removed without invalidating the graph's property of being an almost complete binary tree: the rightmost vertex on the deepest level
- This violates the heap condition (only) at the root. reheap()
- Complexity: $O(d(h)) = O(\log n)$

Algorithm:

key delete_min(heap h){
  v :=root of h
  $k := v.key$
  v' :=rightmost leaf on h's deepest level
  $v.key := v'.key$
  delete v'
  reheap(h)
  return k
}

1.1.2 The delete_min-Operation

- We need to delete a key of minimum value, stored at the root
- We replace the root by the only vertex that can be removed without invalidating the graph's property of being an almost complete binary tree: the rightmost vertex on the deepest level
- This violates the heap condition (only) at the root. reheap()
- Complexity: $O(d(h)) = O(\log n)$

Algorithm:

key delete_min(heap h){
  v :=root of h
  $k := v.key$
  v' :=rightmost leaf on h's deepest level
  $v.key := v'.key$
  delete v'
  reheap(h)
  return k
}

1.1.2 The delete_min-Operation

- We need to delete a key of minimum value, stored at the root
- We replace the root by the only vertex that can be removed without invalidating the graph's property of being an almost complete binary tree: the rightmost vertex on the deepest level
- This violates the heap condition (only) at the root. reheap()
- Complexity: $O(d(h)) = O(\log n)$

Algorithm:

key delete_min(heap h){
  v :=root of h
  $k := v.key$
  v' :=rightmost leaf on h's deepest level
  $v.key := v'.key$
  delete v'
  reheap(h)
  return k
}

1.1.3 The create_heap-Operation

- Given an array of key values, we want to create a heap containing exactly these keys
- We first store them in an almost complete binary tree, assigning the keys to vertices randomly
- To satisfy the heap condition in all vertices, we rearrange the assignment. A method can be derived by induction.

- The approach resulting from this induction merely amounts to applying reheap() to all vertices in the tree, in a bottom-up order.

1.1.3 The create_heap-Operation

- Given an array of key values, we want to create a heap containing exactly these keys
- We first store them in an almost complete binary tree, assigning the keys to vertices randomly
- To satisfy the heap condition in all vertices, we rearrange the assignment. A method can be derived by induction.
- The approach resulting from this induction merely amounts to applying reheap() to all vertices in the tree, in a bottom-up order.

1.1.3 The create_heap-Operation

- Given an array of key values, we want to create a heap containing exactly these keys
- We first store them in an almost complete binary tree, assigning the keys to vertices randomly
- To satisfy the heap condition in all vertices, we rearrange the assignment. A method can be derived by induction.
  - I. The leafs already satisfy the heap condition.
  - II. Suppose, all subtrees on the level $\ell + 1$ satisfy the heap condition. Let v be a vertex on the level ℓ. Then v's subtree violates the heap condition only at its root. Apply reheap().
- The approach resulting from this induction merely amounts to applying reheap() to all vertices in the tree, in a bottom-up order.

1.1.3 The create_heap-Operation

- Given an array of key values, we want to create a heap containing exactly these keys
- We first store them in an almost complete binary tree, assigning the keys to vertices randomly
- To satisfy the heap condition in all vertices, we rearrange the assignment. A method can be derived by induction.
    I. The leafs already satisfy the heap condition.
    II. Suppose, all subtrees on the level $\ell + 1$ satisfy the heap condition. Let v be a vertex on the level ℓ. Then v's subtree violates the heap condition only at its root. Apply reheap().
- The approach resulting from this induction merely amounts to applying reheap() to all vertices in the tree, in a bottom-up order.

1.1.3 The create_heap-Operation

- Given an array of key values, we want to create a heap containing exactly these keys
- We first store them in an almost complete binary tree, assigning the keys to vertices randomly
- To satisfy the heap condition in all vertices, we rearrange the assignment. A method can be derived by induction.
    I. The leafs already satisfy the heap condition.
    II. Suppose, all subtrees on the level $\ell + 1$ satisfy the heap condition. Let v be a vertex on the level ℓ. Then v's subtree violates the heap condition only at its root. Apply reheap().
- The approach resulting from this induction merely amounts to applying reheap() to all vertices in the tree, in a bottom-up order.

1.1.3 The create_heap-Operation

- Given an array of key values, we want to create a heap containing exactly these keys
- We first store them in an almost complete binary tree, assigning the keys to vertices randomly
- To satisfy the heap condition in all vertices, we rearrange the assignment. A method can be derived by induction.
  - I. The leafs already satisfy the heap condition.
  - II. Suppose, all subtrees on the level $\ell + 1$ satisfy the heap condition. Let v be a vertex on the level ℓ. Then v's subtree violates the heap condition only at its root. Apply reheap().
- The approach resulting from this induction merely amounts to applying reheap() to all vertices in the tree, in a bottom-up order.

Algorithm:

heap create_heap(key $A[]$, unsigned n){
  construct an almost complete binary tree h containing the n keys in $A[]$
  for $\ell := d(h)$ downto 1 do
    foreach node v on level ℓ in h do
      t :=subtree rooted at v
      reheap(t)
    od
  od
  return h
}

Complexity: An upper bound for the running time can be obtained easily — We apply reheap n times, hence the complexity is bounded by $O(n \log n)$. In fact, this is not a tight bound, as we shall see shortly.

1.2 Building the HeapSort Algorithm

- We can now take advantage of the Heap data structure with all its operations.
- Using it for sorting an array of keys is simple:

Algorithm:

void HeapSort(key $A[]$, unsigned n){
  heap h :=create_heap(A, n)
  for $i := 1$ to n do
    $A[i]$:=delete_min(h)
  od
}

1.2 Building the HeapSort Algorithm

- We can now take advantage of the Heap data structure with all its operations.
- Using it for sorting an array of keys is simple:
    1. Create a heap containing all the keys
    2. Until the heap is empty, remove the key of minimum value. This results in an increasing sequence.
    3. Store these keys in the array

Algorithm:

void HeapSort(key $A[]$, unsigned n){
  heap h :=create_heap(A, n)
  for $i := 1$ to n do
    $A[i]$:=delete_min(h)
  od
}

1.2 Building the HeapSort Algorithm

- We can now take advantage of the Heap data structure with all its operations.
- Using it for sorting an array of keys is simple:
  1. Create a heap containing all the keys
  2. Until the heap is empty, remove the key of minimum value. This results in an increasing sequence.
  3. Store these keys in the array

Algorithm:

void HeapSort(key $A[]$, unsigned n){
  heap h :=create_heap(A, n)
  for $i := 1$ to n do
    $A[i]$:=delete_min(h)
  od
}

1.2 Building the HeapSort Algorithm

- We can now take advantage of the Heap data structure with all its operations.
- Using it for sorting an array of keys is simple:
  1. Create a heap containing all the keys
  2. Until the heap is empty, remove the key of minimum value. This results in an increasing sequence.
  3. Store these keys in the array

Algorithm:

void HeapSort(key $A[]$, unsigned n)$\{$
  heap h :=create_heap(A, n)
  for $i := 1$ to n do
    $A[i]$:=delete_min(h)
  od
$\}$

1.2 Building the HeapSort Algorithm

- We can now take advantage of the Heap data structure with all its operations.
- Using it for sorting an array of keys is simple:
    1. Create a heap containing all the keys
    2. Until the heap is empty, remove the key of minimum value. This results in an increasing sequence.
    3. Store these keys in the array

Algorithm:

void HeapSort(key $A[]$, unsigned n)$\{$
  heap h :=create_heap(A, n)
  for $i := 1$ to n do
    $A[i]$:=delete_min(h)
  od
$\}$

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
  - The keys are stored in a top-down, left-to-right order.
  - Suppose, a vertex v (or its key) is stored at index i in the array. Then,
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
  - i. the left child of v has index $2i$
  - ii. the right child of v has index $2i + 1$
  - iii. the father of v has index $\lfloor i/2 \rfloor$
  - iv. the level of v is $\ell(v) = \lfloor \log i \rfloor + 1$
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
  i. the left child of v has index $2i$
  ii. the right child of v has index $2i + 1$
  iii. the father of v has index $\lfloor i/2 \rfloor$
  iv. the level of v is $\ell(v) = \lfloor \log i \rfloor + 1$
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
  i. the left child of v has index $2i$
  ii. the right child of v has index $2i + 1$
  iii. the father of v has index $\lfloor i/2 \rfloor$
  iv. the level of v is $\ell(v) = \lfloor \log i \rfloor + 1$
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
  i. the left child of v has index $2i$
  ii. the right child of v has index $2i + 1$
  iii. the father of v has index $\lfloor i/2 \rfloor$
  iv. the level of v is $\ell(v) = \lfloor \log i \rfloor + 1$
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
  i. the left child of v has index $2i$
  ii. the right child of v has index $2i + 1$
  iii. the father of v has index $\lfloor i/2 \rfloor$
  iv. the level of v is $\ell(v) = \lfloor \log i \rfloor + 1$
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
  i. the left child of v has index $2i$
  ii. the right child of v has index $2i+1$
  iii. the father of v has index $\lfloor i/2 \rfloor$
  iv. the level of v is $\ell(v) = \lfloor \log i \rfloor + 1$
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.3 Implementation

- We have not yet seen how to store the heap.
- We could of course use structs and pointers to represent the heap, but thanks to its regular structure there is a better way.
- The keys contained in the heap are stored linearly in an array, while preserving all topological information:
- The keys are stored in a top-down, left-to-right order.
- Suppose, a vertex v (or its key) is stored at index i in the array. Then,
  - i. the left child of v has index $2i$
  - ii. the right child of v has index $2i + 1$
  - iii. the father of v has index $\lfloor i/2 \rfloor$
  - iv. the level of v is $\ell(v) = \lfloor \log i \rfloor + 1$
- Try to prove the above statements as a homwork.
- Try to write down the heap operations in terms of the linearized heap representation.

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:
  - The left hand side contains the heap
  - The right hand side contains a sorted sequence of keys.
  - Initially the right hand side region is empty, and the entire array contains the heap
  - As the algorithm progresses, minimum keys are moved from the heap region into the sorted region
  - Each step requires a delete_min() and a reheap() operation
  - The procedure terminates when the heap region is empty and all keys are sorted

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:
  - The left hand side contains the heap
  - The right hand side contains a sorted sequence of keys.
  - Initially the right hand side region is empty, and the entire array contains the heap
  - As the algorithm progresses, minimum keys are moved from the heap region into the sorted region
  - Each step requires a delete_min() and a reheap() operation
  - The procedure terminates when the heap region is empty and all keys are sorted

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:
  - The left hand side contains the heap
  - The right hand side contains a sorted sequence of keys.
  - Initially the right hand side region is empty, and the entire array contains the heap
  - As the algorithm progresses, minimum keys are moved from the heap region into the sorted region
  - Each step requires a delete_min() and a reheap() operation
  - The procedure terminates when the heap region is empty and all keys are sorted

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:
  - The left hand side contains the heap
  - The right hand side contains a sorted sequence of keys.
  - Initially the right hand side region is empty, and the entire array contains the heap
  - As the algorithm progresses, minimum keys are moved from the heap region into the sorted region
  - Each step requires a delete_min() and a reheap() operation
  - The procedure terminates when the heap region is empty and all keys are sorted

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:
  - The left hand side contains the heap
  - The right hand side contains a sorted sequence of keys.
  - Initially the right hand side region is empty, and the entire array contains the heap
  - As the algorithm progresses, minimum keys are moved from the heap region into the sorted region
  - Each step requires a delete_min() and a reheap() operation
  - The procedure terminates when the heap region is empty and all keys are sorted

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:
  - The left hand side contains the heap
  - The right hand side contains a sorted sequence of keys.
  - Initially the right hand side region is empty, and the entire array contains the heap
  - As the algorithm progresses, minimum keys are moved from the heap region into the sorted region
  - Each step requires a delete_min() and a reheap() operation
  - The procedure terminates when the heap region is empty and all keys are sorted

1.4 Putting all of it together

- We use the linearized representation for heaps: Key array $A[]$ is used to contain the heap structure.
- $A[]$ is subdivided into two regions:
    - The left hand side contains the heap
    - The right hand side contains a sorted sequence of keys.
    - Initially the right hand side region is empty, and the entire array contains the heap
    - As the algorithm progresses, minimum keys are moved from the heap region into the sorted region
    - Each step requires a delete_min() and a reheap() operation
    - The procedure terminates when the heap region is empty and all keys are sorted

Algorithm:

```
void HeapSort(key A[], unsigned n){
  for k := n downto 1 do // create_heap
    reheap(A, n, k)
  od
  for k := n downto 1 do // n× delete_min
    swap A[1] and A[k]
    reheap(A, k, 1)
  od
  for k := 1 to ⌊n/2⌋ do // reverse sorted array
    swap A[k] and A[n − k + 1]
  od
}
```

reheap:

```
void reheap (key A[], unsigned n, unsigned r){// r ≡ root
   unsigned i := r // current node
   unsigned j := 2r // 1st child of i
   while (j ≤ n) do
      if (j + 1 ≤ n ∧ A[j + 1] < A[j]) then j + + fi // use 2nd child
      if (A[j] < A[i]) then
         swap A[i] and A[j]
         j:=2j
      else break// heap condition satisfied
      fi
   od
}
```

1.5 Complexity

Let us analyze the number of key comparisons needed to sort n keys, using MergeSort.

To this end, we define
$V_{\text{reheap}}(n,i) := \#$ comparisons for reheap() in subtree rooted at i

In the worst case, reheap() has to descend all the way from the root to a leaf. Hence, it holds that

$$V_{\text{reheap}}(n,i) \leq 2 \cdot (\lfloor \log n \rfloor - \lfloor \log i \rfloor)$$

1.5 Complexity

Let us analyze the number of key comparisons needed to sort n keys, using MergeSort.

To this end, we define
$V_{\text{reheap}}(n, i) :=$ # comparisons for reheap() in subtree rooted at i

In the worst case, reheap() has to descend all the way from the root to a leaf. Hence, it holds that

$$V_{\text{reheap}}(n, i) \leq 2 \cdot (\lfloor \log n \rfloor - \lfloor \log i \rfloor)$$

1.5 Complexity

Let us analyze the number of key comparisons needed to sort n keys, using MergeSort.

To this end, we define
$V_{\text{reheap}}(n, i) :=$ # comparisons for reheap() in subtree rooted at i

In the worst case, reheap() has to descend all the way from the root to a leaf. Hence, it holds that

$$V_{\text{reheap}}(n, i) \leq 2 \cdot (\lfloor \log n \rfloor - \lfloor \log i \rfloor)$$

For create_heap() we define

$V_{\text{create}}(n)$:= # comparisons for create_heap()

Here it holds that

$$
\begin{aligned}
V_{\text{create}}(n) &\leq \sum_{i=1}^{n} V_{\text{reheap}}(n,i) \\
&\leq 2\sum_{i=1}^{n}(\lfloor \log n \rfloor - \lfloor \log i \rfloor) \\
&\leq 2\sum_{i=1}^{n}(\log n - \log i + 1) \\
&= 2n\log n + 2n - 2\sum_{i=2}^{n}\log i \\
&\leq^* 2n\log n + 2n - 2n\log n - 2/\ln 2(n-1) \\
&\leq 5n
\end{aligned}
$$

This shows that our upper bound on the complexity of create_heap was too pessimistic. $*$ It holds that

$2\sum_{i=2}^{n}\log i \geq 2n\log n - 2/\ln 2(n-1)$

For create_heap() we define
$V_{\text{create}}(n) :=$ # comparisons for create_heap()

Here it holds that

$$
\begin{aligned}
V_{\text{create}}(n) &\leq \sum_{i=1}^{n} V_{\text{reheap}}(n,i) \\
&\leq 2\sum_{i=1}^{n} (\lfloor \log n \rfloor - \lfloor \log i \rfloor) \\
&\leq 2\sum_{i=1}^{n} (\log n - \log i + 1) \\
&= 2n \log n + 2n - 2\sum_{i=2}^{n} \log i \\
&\leq^{*} 2n \log n + 2n - 2n \log n - 2/\ln 2(n-1) \\
&\leq 5n
\end{aligned}
$$

This shows that our upper bound on the complexity of create_heap was too pessimistic. $*$ It holds that

$$2\sum_{i=2}^{n} \log i \geq 2n \log n - 2/\ln 2(n-1)$$

Finally, we define

$V_{\text{sort}}(n) :=$ # comparisons for the sorting procedure.

Here we have

$$V_{\text{sort}}(n) = \sum_{k=1}^{n} V_{\text{reheap}}(k, 1) \leq 2n \log n$$

In total, the number of comparisons for HeapSort is

$$V_{\text{create}}(n) + V_{\text{sort}}(n) = O(n \log n).$$

Finally, we define
$V_{\text{sort}}(n) :=$ # comparisons for the sorting procedure.

Here we have

$$V_{\text{sort}}(n) = \sum_{k=1}^{n} V_{\text{reheap}}(k, 1) \leq 2n \log n$$

In total, the number of comparisons for HeapSort is

$$V_{\text{create}}(n) + V_{\text{sort}}(n) = O(n \log n).$$