Masaryk University
Faculty of Informatics

Priority queues

Bachelor's Thesis

Marek Chocholáček

Brno, Spring 2019

Masaryk University
Faculty of Informatics

Priority queues

Bachelor's Thesis

Marek Chocholáček

Brno, Spring 2019

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marek Chocholáček

Advisor: prof. RNDr. Ivana Černá Csc

Acknowledgements

I want to thank prof. Černá for provided insight and help. I also want to thank Maroš Beťko and Matúš Kiša that helped me proofread this thesis.

Abstract

In my thesis I am implementing and experimentally testing several heap variants commonly used as priority queues. These variants include binary heap, Fibonacci heap, binomial heap, violation heap and rank-pairing heap. I am testing these structures on both artificial and real-world workloads.

Keywords

priority queue, heap, binary heap, implicit binary heap, explicit binary heap, binomial heap, Fibonacci heap, violation heap, rank-pairing heap, benchmark, graph

Contents

List of Tables

List of Figures

Introduction

Priority queues are, in practice, a widely used and well-known term in computer science community. To this day, many theoretical variants supporting different sets of operations with different guarantees have been proposed. In my thesis, I focus only on heaps supporting these operations:

- $Empty()$ - indicates if any elements are stored within the heap
- $x\ Min()$ - returns handle x to the element with the minimum key. Exception is thrown if the structure is empty.
- $x\ Insert(k,\ i)$ - inserts the item i with the key k into the heap and returns handle x to inserted element
- $x\ ExtractMin()$ - delete the item with the minimum key from the heap, update the heap and return a handle to deleted item x.
- $DecreaseKey(x,\ y)$ - given a handle x, change the key and update the heap.

It is well known that either $Insert$ or $DeleteMin$ must take $\Omega(\log n)$ time due to the classic lower bound for sorting [1]. Other operations can be done in $O(1)$ time. Worst case times are in practice rarely encountered or can be treated as constant, which leads to preference of standard simple structures instead of their more complicated constant-time alternatives [2].

Algorithms I implemented are based on algorithms proposed in other papers [3, 4, 5, 6, 7, 8]. As implementation language I chose C++, solely because of performance concerns. C++ being a low level language, a pointer is chosen as the handle type of every implemented structure. This choice required minor changes to some of the structures and will be explained further in the following chapter. In chapter 1 I provide simplified pseudocodes to every implementation-important function. All of the implementations are public and can be found on my github[1].

1. `https://github.com/chocholacek/Priority-queues/tree/master/src`

1 Heap variants

Heap variants I implemented differ not only in logic lines of code but also in the complexity of their operations as is shown in table 1.1. In this chapter, I would like to point out some of the major differences between the heap variants I implemented and provide insight on how their operations work. Keep in mind that in pseudocodes shown in this chapter I use array indexing from zero and asymptotic time bounds of heap operations are taken from other sources [2, 3, 4, 8, 5, 7] and are not proven by me.

Table 1.1: Amortized time of heap operations

	Min	Insert	DecreaseKey	ExtractMin
binary	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
binomial	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Fibonacci	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
violation	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
rank-pairing	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$

Payloads are stored in the node structure and each heap implements its own variant of this structure. By default this structure contains a key, an item and can be constructed as $Node(k,\ i)$ with the given key k and the item i.

```
1 struct Node:
2   int key
3   Item item
```

I also set the prerequisite that payload stays in the same node through the node's lifetime in the heap. This choice leads to minor changes in some of the heap variants and these changes are covered in the section of every heap. These changes do not affect amortized asymptotic time bounds of any implemented heap.

1.1 Binary heap

From all of the possible binary heap implementations I am only considering and implementing those that store elements implicitly in an array or explicitly in a tree structure. Implementation differences between these are two shown in the corresponding subsections 1.1.1 and 1.1.2. As shown in table 1.3, these heaps do not differ in time bounds of their operations.

Table 1.2: Comparison of binary heap implementations

Operation	implicit binary heap	explicit binary heap
Min	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(\log n)$
ExtractMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(\log n)$

1.1.1 Implicit binary heap

By definition, implicit binary heap stores elements in an allocated array and the key-item pairs are swapped when needed. As mentioned before, I chose a pointer as the handle for every structure, therefore I needed to find a workaround for swapping payloads, since the payload will not remain in the same node through the node's lifetime. Keys and items are stored within the node structure defined below.

```
1 struct Node:
2   int key
3   Item item
4   int index
```

The implicit binary heap itself stores an array to these allocated nodes.

```
1 struct ImplicitBinaryHeap:
2   Node[] array
```

Finding the node with the minimum key stored within the structure is done in $O(1)$ time [3] as the node containing the minimum key is stored on index 0 of the underlying array.

DecreaseKey swaps the node upwards if its key is smaller than that of its parent until the minimum-heap property is preserved or the root is reached, therefore time complexity of this operation is $O(\log n)$ on an array of size n [3]. Index of the node x that is provided to *DecreaseKey* is stored within the x so finding node x in the array is not required.

```
1  DecreaseKey(x, k):
2    x = array[i]
3    if k > x.key:
4      error
5
6    x.key = k
7    while x has parent and its key is smaller:
8      swap x and its parent in array
9      swap their stored indices
10     x = parent of x
```

New element is inserted by creating a new node with the *key* property set to infinity and calling *DecreaseKey* with the provided key k. In my implementation, I use max value of an integer as a representation of positive infinity.

```
1  Insert(k, i):
2    n = new Node(∞, i)
3    n.index = size of the array
4    insert n to the back of the array
5    DecreaseKey(n, k)
```

Extracting a node with the minimum key is done by swapping pointers on the first and the last index of the array, then the formerly first node is removed from the array and $HeapifyDown$ is called on the newly set first node.

```
1  ExtractMin():
2    if (Empty()):
3      error
4    min = the first node of the array
5    last = the last of node of the array
6    swap min and last in the array
7    swap indices of min and last
8    remove last node from the array
9    HeapifyDown(0)
10   return min
11
12 HeapifyDown(i):
13   if i is out of bounds:
14     return
15   x = array[i]
16   find child s with smallest key
17   if s exists and s.key < x.key:
18     swap s and x in the array
19     swap indices of x and s
20     HeapifyDown(s.index)
```

The Swapping of the nodes is done in $\Theta(1)$ time and because of the heap's tree-like form, *HeapifyDown* finishes at worst in logarithmic time. The time complexity of *ExtractMin* is therefore the same, i.e. $O(\log n)$.[3]

The parent and sibling indices are calculated from the index i in $\Theta(1)$ time and are given by:

$$\begin{aligned} Parent(i) &= (i-1)/2 \\ Left(i) &= i*2+1 \\ Right(i) &= i*2+2 \end{aligned}$$

1.1.2 Explicit binary heap

The distinction between the implicit binary heap and an explicit binary heap comes down to element storage. The explicit binary heaps stores elements in tree-like structure, which is very similar to the binary

search tree [3], with nodes containing pointers to the child nodes and pointer to the parent node.

```
1 struct Node:
2   int key
3   Item item
4   Node* parent
5   Node* left
6   Node* right
```

My implementation of the explicit binary heap is storing a pointer to the root node of the tree and a pointer to the last inserted node, which is used for finding a place for an new node when insertion into the heap takes place. Finding a node with the minimum key is constant since it is always stored in the root pointer.

```
1 struct ExplicitBinaryHeap:
2   Node* root
3   Node* last
```

Before diving into the *Insert* function, I need to point out how exactly a place for a new node is found. While the last node is the right child of its parent, traverse the tree upwards. If root was not reached, either the right sibling does not exist, meaning a place for the new node was found and execution is stopped or continue by moving to the right sibling. Then traverse down to the left as much as possible to find a place for the new node if a place is not known yet. Since the worst case scenario is that the tree is traversed from the rightmost to the leftmost leaf of the tree, this operation has $O(\log n)$ complexity [6]. I refer to this function as *InsertChild* and it is used only by the *Insert* function.

Whenever new key-item pair is being inserted to the tree, two situations may occur. Either the tree is empty and the new node is inserted as the root or the new node is inserted as a child. If the second case arises,the minimum-heap property must be regained by swapping nodes upwards.

```
1  Insert(k, i):
2    n = new Node(key, item)
3    if Empty():
4      root = last = n
5    else:
6      InsertChild(n)
7      HeapifyUp(n)
8    return n
9
10 InsertChild(n):
11   cur = last
12   while cur is right child:
13     cur = cur.parent
14
15   if cur != root:
16     if cur.parent.right exists:
17       n.parent = cur.parent
18       cur.parent.right = n
19       last = cur.parent.right
20       return
21     cur = cur.parent.right;
22
23   while cur.left exists:
24     cur = cur.left
25
26   n.parent = cur
27   cur.left = n
28   last = cur.left
```

HeapifyUp in the explicit binary heap works similarly as the *Heapify* described before in its implicit counterpart with an exception regarding the node swapping. My prerequisite being that once an item gets inserted into the heap and the corresponding pointer to the node containing it is returned, the item itself cannot change. Therefore swapping payloads between nodes would not be sufficient and I had to swap entire nodes. Even though this operation has a lot higher operation count than its counterpart in the implicit binary heap, it still takes $O(1)$ time.

```
1  Swap(up, lo):
2    swap child pointers of up and lo
3    lo.parent = up.parent
4    up.parent = lo
5    for each child c of up that exists:
6      c.parent = up
7    if lo has parent:
8      if left sibling of lo == up:
9        lo.parent.left = lo
10     else:
11       lo.parent.right = lo
12   else:
13     root = lo
14   if lo == lo.left
15     lo.left = up
16   else:
17     lo.right = up
18   for each child c of lo that exists:
19     c.parent = lo
20   if lo is last:
21     last = up
22   return lo
```

```
1  HeapifyUp(n):
2    while n has parent and n.key < n.parent.key:
3      n = Swap(n.parent, n)
```

In the worst case, $HeapifyUp$ traverses from leaf to the root, having time complexity of $O(\log n)$. This makes *Insert* run at worst in $O(\log n)$ time.

Extracting a node with the minimum key is theoretically identical to *ExtractMin* in the implicit binary heap. The root node is swapped with the last node in the tree, the last node is removed from the tree, $HeapifyDown$ starts from the root and the removed node is returned. *ExtractMin* utilizes two logarithmic functions, namely *DeleteMin* and $HeapifyDown$ mentioned before, thus its complexity is $O(\log n)$.

```
1  ExtractMin():
2    if Empty():
3      error
4    r = root
5    swap child pointers of r and last
6    for each child c of last that exists:
7      c.parent = last
8    swap parent pointers of r and last
9    if r.parent exists:
10     if r.parent.left is last:
11       r.parent.left = r
12     else
13       r.parent.right = r
14   root = last
15   last = r
16   ret = DeleteLast()
17   HeapifyDown(root)
18   return ret
19
20 HeapifyDown(n):
21   find child s with smallest key
22   if s exists and s.key < n.key:
23     Swap(n, s)
24     HeapifyDown(n)
```

DeleteLast function removes the node pointed to by the last pointer preserved within the heap structure and reassigns another node as new last node. This is done in similar fashion as it was done in *InsertChild*, only the process is reversed. Additionally, this process takes $O(\log n)$ as well [6].

```
1  DeleteLast():
2    cur = last
3    ret = last
4    if cur is root:
5      root = last = null
6      return ret
7    while cur is left child:
8      cur = cur.parent
9    if cur is not root:
10     cur = cur.parent.left
11   while cur.right exists:
12     cur = cur.right
13   remove last from its parent
14   last = cur
15   return ret
```

The only remaining operation is *DecreaseKey*. After the key is changed $HeapifyUp$ is called, which makes *DecreaseKey* run in $O(\log n)$ time.

```
1  DecreaseKey(x, k):
2    if k > x.key
3      error
4    x.key = k
5    HeapifyUp(x)
```

1.2 Binomial heap

A binomial heap is implemented as a set of binomial trees [7]. Each of these trees obeys the property of the minimum heap. Only one or zero trees of the same degree can be present in the structure. In addition to a key and item, the node contains a list of children.

```
1  struct Node:
2    int key
3    Item item
4    Node[] children
```

Note that degree of the node can be calculated from the length of the

children list. The binomial heap itself stores the roots of the binomial trees in a list and the total number of the roots in the structure in stored in property *count*.

```
1 struct BinomialHeap:
2   Node[] roots
3   int count
```

Complexity-wise, the main difference between the binomial and the binary heap is that finding the minimum is done in $O(\log n)$ time [7] compared to the $O(1)$ time [3] of the binary heap. However, it is shown that *Insert* in the binomial heap can run in $O(1)$ amortized time compared to the $O(\log n)$ run-time of the binary heap [7].

Table 1.3: Comparison of the binary and the binomial heap

Operation	binary heap	binomial heap (amortized)
Min	$O(1)$	$O(logn)$
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(\log n)$

A node with the minimum key is found in the root list, because every binomial tree obeys minimum-heap property. Given this fact, *Min* needs to loop only through the root list and return node with minimum key.

```
1 Min():
2   if Empty():
3     error
4   n = first element in the root list
5   for each root r in the root list:
6     if r.key < n.key:
7       n = r
8   return n
```

As mentioned before, the root list can contain a maximum of one tree of the same degree, which means consolidation of the structure is needed whenever a new node is inserted into the root list. This is done by merging the trees with the same degree. Trees x and y are

merged by making the tree with the larger key a child of the tree with the lesser key.

```
1  Merge(x, y):
2    if y.key < x.key
3      swap x and y
4    y.parent = x
5    insert y to the children list of x
6    return x
```

Consolidating the structure is done by looping through the root list, finding two roots with the same degree, merging them and storing the result in the temporary array. Length of the temporary array is dependent on the maximum degree md of the tree that can be merged from c nodes, where c is the total number of nodes in the structure. Maximum degree is calculated by:

$$md = \lceil log_2(c) \rceil + 1 \tag{1.1}$$

Index of the node x in the temporary array is given by x's degree. If another node y is found on index of x, x and y are merged and inserted at increased index. This is repeated until an empty place is found. After all of the roots were inserted to the temporary array, the root list is cleared and the nodes stored in the temporary array are re-inserted to the root list.

```
1  Consolidate():
2    arr = allocate array with size of max degree
3    for each root x in the root list:
4      d = x.degree
5      while arr[d] != null:
6        y = arr[d]
7        arr[d] = null
8        x = Merge(x, y)
9        d = x.degree
10     arr[d] = x
11   clear root list
12   for each node r in arr that is not null:
13     insert r to the root list
```

After *Merge* and *Consolidate* were defined, inserting a new node n into the heap is as easy as adding n to the root list and calling *Consolidate*.

```
1 Insert(k, i):
2   n = new Node(k, i)
3   add n to the root list
4   Consolidate()
5   return n
```

Extraction of the node with the minimum key needs to perform a search for such node. This is done by calling *Min* and getting the node n with the minimum key in $O(\log n)$ time. Children of n are then promoted to the root list, n is removed from the root list and the structure is consolidated.

```
1 ExtractMin():
2   n = Min()
3   promote children of n to the root list
4   remove n from the root list
5   Consolidate()
6   return min
```

Decreasing the key of the node x in the binomial heap is done by decreasing its key and recursively swapping the node upwards until the minimum-heap property is preserved. Again, instead of swapping payloads I opted out to swapping entire nodes. This is done to ensure the prerequisite of the node having the same payload through its lifetime in the heap.

```
1 DecreaseKey(x, k):
2   if k > x.key:
3     error
4   x.key = k
5   while x has parent and its key is smaller:
6     Swap(x.parent, x)
```

Swapping of the nodes is done in the *Swap* function and is shown below.

```
1  Swap(x, y):
2    cxi =  position of y in x children list
3    y.parent = x.parent
4    if y.parent != null:
5      cyi = position of x in y.parent.children
6      y.parent.children[cyi] = y
7    else:
8      it = find x in the root list
9      it = y
10   x.parent = y
11   x.children[cxi] = x
12   for every child c of x:
13     c.parent = y
14   for every child c of y:
15     c.parent = x
16   swap x.children and y.children
```

1.3 Fibonacci heap

A Fibonacci heap structure is one of the more complicated structures I implemented. Its advantage is in several of its operations running in constant amortized time, which makes this structure well suited for application that invoke these operations frequently. [3] Similarly to the binomial heap, the Fibonacci heap stores binomial trees but the prerequisite of having a maximum of one tree with the same degree is not enforced in this structure. As table 1.4 shows this should bring significant improvements to the run-time speeds.

Table 1.4: Comparison of the binomial and the Fibonnaci heap

Operation	binomial (amortized)	Fibonacci (amortized)
Min	$O(\log n)$	$O(1)$
Insert	$O(1)$	$O(1)$
ExtractMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(1)$

Even though instruction count is significantly higher in the Fibonacci heap due to more complex implementation, the Fibonacci heap can outperform simpler heaps. As table 1.4 shows, the difference in run-time can be significant by calling the right set of operations. In fact, some of the graph algorithms, e.g. Dijkstra's algorithm, can use *DecreaseKey* once per edge, which can lead to significant performance gain. In theory, the Fibonacci heap should easily outperform the simpler implementations on dense graphs with many edges where $O(1)$ amortized time for each edge adds up to the significant performance difference over $O(\log n)$ of the binary heaps [3].

Programming complexity of the Fibonacci heap makes them less desirable in practice than the more simple and straightforward implementations of the binary heap. The Fibonacci heap is a collection of rooted node trees that are min-heap ordered [3]. This root collection is kept in a circular doubly-linked list for a constant insert and removal.

Additionally to the key k and item i, a node of this list contains pointer *parent* to parent, two pointers to siblings, pointer *child* to one of the children, *degree* as the number of children and boolean *mark* used for cutting the tree. Children of a node are stored in a circular doubly-linked list as well.

```
1 struct Node:
2   int key
3   Item item
4   int degree
5   bool mark
6   Node* next
7   Node* prev
8   Node* parent
9   Node* child
```

The Fibonacci heap holds a pointer to the root list pointing to the node with the minimum key so the minimum is found in $\Theta(1)$ time. The number of roots named *RootSize* and the total number of nodes named *Count* are preserved within the heap structure as well.

```
1 struct FibonacciHeap:
2   Node* minimum
3   int RootCount
4   int Count
```

A new instance of the node can be created as $Node(k, i)$, which sets all of the pointers to null. Degree of a newly created node is always 0 and *mark* is always set to false.

Inserting to the heap is done by inserting a new node into the root list. If the heap is empty, *minimum* pointer is set to this new node and no further actions are needed. Otherwise keys of *minimum* and the new node are compared and *minimum* pointer is updated if needed. As noted before, inserting into the circular list takes constant time, therefore *Insert* runs in $O(1)$ time.

```
1 Insert(k, i):
2   n = new Node(k, i)
3   add n to the root list
4   if n.key < minimum.key:
5     minimum = n
```

The operation of extracting the node with the minimum key is the most complicated operation in this section. It consists of multiple complex operations which will be explained in greater detail later in the section. *ExtractMin* runs in $O(\log n)$ time [3]. Extracting starts by testing the heap for emptiness, which may result in terminating the whole process. Then the children of the *minimum* are promoted into the root list and *minimum* itself is removed from the list. Both removing and adding node to the circular doubly-linked list is done in constant time as it only changes *prev* and *next* pointers of the node's siblings. Both *RootSize* and *Count* decrease by one and if *RootSize* is equal to zero, *minimum* is set to null. Otherwise, *minimum* is set to its next sibling and the structure is consolidated in such a way that the maximum number of the trees with the same degree is 1.

```
1  ExctractMin():
2    if Empty():
3      error
4    ret = minimum
5    promote children of ret to the root list
6    remove ret from the root list
7    RootSize = RootsSize - 1
8    if RootSize == 0:
9      minimum = null
10   else:
11     minimum = minimum.next
12     Consolidate()
13   Count = Count - 1
14   return ret
```

Consolidation of the structure begins with allocating an array, size of which is equal to the maximum possible degree. Similarly to the binomial heap, the greatest possible degree depends on the total count of the nodes in the structure and is calculated by formula 1.1. Each element in the allocated array is then set to null and the process continues by visiting every root x in the root list. For each x, the following is done until every root has a distinct degree:

1. Find root y where y.degree == x.degree.
2. If y.key < x. key swap x and y.
3. Link x and y if y exists, in such a manner that a node with the smaller key is parent of a node with the larger key.
4. Increase x.degree by 1

```
1  Consolidate():
2    arr = allocate array with size of max degree
3    set every element of arr to null
4    x = minimum
5    for each root r in the root list:
6      n = x.next
7      d = x.degree
8      while arr[d] != null:
9        y = array[d]
10       if y.key < x.key
11         swap x and y
12       Link(x, y)
13       array[d] = null
14       ++d
15     array[d] = x
16     x = n
17   minimum = null
18   for each root r in arr:
19     insert r to the root list
```

Since every node in the structure is stored in doubly-linked list, linking nodes is simply done by removing a node from one list and inserting it to the other list.

```
1  Link(y, x):
2    remove y from its list
3    add y to the child list of x
4    y.mark = false
```

Next operation is *DecreaseKey* and it runs in amortized time $O(1)$ [3]. New key k is assigned to the node x after validating that k is not larger than the old key. If the minimum-heap property is violated, x is cut from the list and inserted as new root.

```
1  DecreaseKey(x, k):
2    if k > x.key:
3      error
4    x.key = k
5    y = x.parent
6    if y exists and x.key < y.key:
7      Cut(x, y)
8      CascadingCut(y)
9    if x.key < minimum.key:
10     minimum = x
11
12 Cut(x, y):
13   remove x from y.child list
14   decrease y.degree by 1
15   if y.degree == 0:
16     y.child = null
17   add x to the root list
18   x.parent = null
19   x.mark = false
```

As long as the node y loses the second child, it is cut from its parent and inserted as a root. The node's property *mark* is used to ensure such functionality and to obtain the desired time bounds [3]. The cutting process is repeated while y has a parent and y is marked. If unmarked y is reached, it is marked and execution stops. I refer to this operation as *CascadingCut*.

```
1  CascadingCut(y):
2    z = y.parent
3    if z exists:
4      if y.mark == false:
5        y.mark = true
6      else:
7        Cut(y, z)
8        CascadingCut(z)
```

1.4 Violation heap

Similar to the Fibonacci heaps, the violation heap is a set of heap-ordered node-disjoint multiary trees [5]. In addition to the key and item, each node contains two pointers to siblings, a pointer to a child and an integer representing the degree of the node.

```
1 struct Node:
2   int key
3   Item item
4   Node* next
5   Node* prev
6   Node* child
7   int degree
```

These properties are utilized as follows:

- Roots are stored in circular singly-linked forward list stored within the heap structure as pointer to the root with the minimum key. This pointer is referred to as *minimum*
- Children of the node n are ordered accordingly to time when they were inserted with the last-inserted node being first in the list and stored in $n.child$ pointer.
- Given any node n, pointer *prev* of $n.child$ points back to n.

In addition to the root list, the violation heap stores total node count and total size of the root list.

```
1 struct ViolationHeap:
2   Node* minimum
3   int RootCount
4   int Count
```

Node is called active if it is one of the two most recently added child nodes. Degree r of the node is updated whenever degrees r_1 and r_2 of its active children decrease and is given by the formula:

$$r = \lceil (r_1 + r_2)/2 \rceil + 1 \tag{1.2}$$

If a node has one or zero children, the degree of the missing child is -1.

A new node n is inserted to the violation heap in the first position if its key is lesser than *minimum*'s key. If n's key is bigger, n is inserted in the second position instead. Pointers of a new node are always set to null and *degree* is set to 0.

```
1  Insert(k, i):
2    n = new Node(k, i)
3    n.next = n
4    ++Count
5    if minimum == null:
6      minimum = n
7      return n
8    if n.key < minimum.key:
9      n.next = minimum
10     minimum = n
11   else:
12     n.next = minimum.next
13     minimum.next = n
14   return n
```

If a key of node x is to be decreased to key k, $x.key$ is set to k. If x is root, stop. Before stopping make x a new *minimum* if x's key is smaller than *minimum*'s key. If x is the active node whose key is not smaller than its parent, stop. Otherwise, cut the subtree x and glue an active child of x with the biggest degree in its position. Recalculate x's degree using formula 1.2. Insert x to the root list as root and make it a new *minimum* if its key is smaller than *minimum*'s. Propagate degree updates from the old position of x to the ancestors of x as long as the visited node is active and its recalculated degree is smaller than its old degree.

```
1  DecreaseKey(x, k):
2    if k > x.key:
3      error
4    x.key = k
5    if x is root:
6      if x.key < minimum.key:
7        minimum = x
8      return
9    if x is active and heap-order is not violated:
10     return
11   lc = child of x with largest degree
12   if lc exists:
13     replace x by lc
14   else:
15     remove x from child list of its parent
16   recalculate degree of x
17   if x was active:
18     cur = parent of x
19   else:
20     cur = null
21   if cur != null:
22     or = cur.degree
23     recalculate degree of cur
24     while cur.degree < or and cur is active child:
25        or = cur.degree
26        recalculate degree of cur
27        cur = parent of cur
28   x.prev = null
29   x.next = x
30   add x to the root list
31   if x.key < minimum.key:
32     minimum = x
```

To remove a node with a minimum key, remove *minimum* from the root list and make each of its children a root in the root list. Repeatedly 3-way-join [5] trees of equal degree until no three trees of the same degree remain. To 3-way-join roots, the following is done:

1. Find and remove three trees with equal degrees from the root list.
2. Find tree z whose key is the smallest among found trees.
3. From the remaining two trees, z_2 is a tree with the larger degree and z_1 is the remaining tree. This ensure that a node with the bigger degree will be inserted as second.
4. Add z_1 to the child list of z.
5. Add z_2 to the child list of z.
6. Increase degree of z by one.
7. Insert z back to the root list.

In pseudocode, I refer to this operation as *Consolidate*. It is worth noting, that as in the Fibonacci heap this operation runs in $O(1)$ amortized time [5].

```
1  ExtractMin():
2    if Empty():
3      error
4    z = minimum
5    promote children of z to the root list
6    Count = Count - 1
7    if Count == 0:
8      minimum = null
9      return z
10   minimum = minimum.next
11   Consolidate()
12   return z
```

In summary, the violation heap contains less lines of code, stores nodes in more practical way and has the same asymptotic run-time bounds as the Fibonacci heap.

1.5 Rank-pairing heap

A rank-pairing heap is relaxed Fibonacci-like heap that utilizes half-ordered single-elimination tournaments [4]. Payload is stored in the

node structure along with pointers to the left child, pointer to the next sibling, a pointer to the parent and an integer representing the node's degree.

```
1 struct Node:
2   int key
3   Item item
4   Node* left
5   Node* next
6   Node* parent
7   int degree
```

The rank-pairing heap is represented by singly-linked circular list of the half-ordered trees [4], and the root with the smallest key is stored in the pointer named *minimum*. This means no searching for minimum node is required and it can be returned in $O(1)$ time. Only trees of the same degree can be linked, and linking of the nodes x and y is done as follows: compare the keys of x and y. If x and y are nodes of the lesser and greater key, respectively, detach the left subtree of x and store it to the *next* pointer of y; then store y to the left subtree of x. This linking operation takes $O(1)$ time [4].

```
1 Link(x, y):
2   if y == null:
3     return x
4   if y.key < x.key:
5     swap x and y
6   y.parent = x
7   if x.next != null:
8     y.next = x.left
9     y.next.parent = y
10  x.left = y
11  x.degree = y.degree + 1
12  return x
```

In addition to the *minimum* pointer, the rank-pairing heap stores the total count of the nodes in the structure. Since *minimum* pointer is stored, the node with the minimum key is found in $O(1)$.

```
1 struct rankpairingHeap:
2   Node* minimum
3   int count
```

Inserting a new node to the rank-pairing heap is done by creating a new node with *degree* set to 0 and inserting it to the root list of the heap. The new node is inserted to the first position if its key is smaller than *minimum*.key, otherwise in the second position. Inserting a new node takes $O(1)$ time [4], as insert to the circular link list takes $O(1)$ time.

```
1 Insert(k, i):
2   n = new Node(k ,i)
3   add n to roots
4   count  = count + 1
5   return n
```

ExtractMin of the rank-pairing heap works similarly to the *ExtractMin* of the Fibonacci heap. First, array b with size of the maximum possible degree is allocated. Similar to the binomial heap, the maximum possible degree of any stored tree can be calculated from the total number of the nodes in the structure and is given by formula 1.1. Every child node in the left subtree is merged into b by calling *Link* on the two nodes with the same degree. Amortized cost of *ExtractMin* is $O(1)$ [4].

```
1 ExtractMin():
2   if Empty():
3     error
4   m = minimum
5   b = allocate array with size of max degree
6   merge every child of m to the b
7   m.left = null
8   merge every root in the root list to b
9   minimum = null
10  for each node n in b:
11    add n to the root list
12  return m
```

Decreasing the key of the node x to k starts by assigning k to $x.key$. If x is root before stopping execution, update *minimum* if the new key of x is lesser than the key of *minimum*. If x is not root, the process continues as follows:

1. x is removed from the parent, and x's *next* and *parent* are set to null.
2. Degree of x is recalculated.
3. x is added to the root list.
4. Degree of x's parent is reduced by type-1 or type-2 degree reduction.

To reduce degree, let $p = x.parent$ and repeat until it stops:

- Type-1 degree reduction:
  If p is a root, set $p.degree = p.left.degree + 1$ and stop. Otherwise, let $v = p.left$ and $w = p.next$. Calculate k by 1.3. If $k \geq p.degree$, stop. Otherwise, let $p.degree = k$ and $p = p.parent$. [4]

$$k = \begin{cases} max(v.degree,\ w.degree) & \text{if } v.degree \neq w.degree \\ v.degree + 1 & \text{otherwise} \end{cases} \tag{1.3}$$

- Type-2 degree reduction:
  If p is a root, set $p.degree = p.left.degree + 1$ and stop. Otherwise, let $v = p.left$ and $w = p.next$. Calculate k by 1.4. If $k \geq p.degree$, stop. Otherwise, let $p.degree = k$ and $p = p.parent$. [4]

$$k = \begin{cases} max(v.degree,\ w.degree) & \text{if } |\text{v.degree - w.degree}| > 1 \\ max(v.degree, w.degree) + 1 & \text{otherwise} \end{cases} \tag{1.4}$$

The rank-pairing heap being relaxed version of the Fibonacci heap [4], this operation falls under the same amortized time bounds of $O(1)$ as *DecreaseKey* in the Fibonacci heap [4], with type-2 degree reduction having smaller constants in the time bounds than the type-1 degree reduction.

```
1  DecreaseKey(x, k):
2    if k > x.key:
3      error
4    x.key = k
5    if x is in the root list:
6      if x.key < minimum.key:
7        minimum = x
8      return
9    p = x.parent
10   if x is left child:
11     p.left = x.next
12     if p.left != null:
13       p.left.parent = p
14   else:
15     p.next = x.next
16     if p.next != null:
17       p.next.parent = p
18   x.next = x.parent = null
19   recalculate degree of x
20   add x to the root list
21   if p is root in the root list:
22     recalculate degree of p
23   else:
24     reduce degrees of p
```

2 Experiments

As stated earlier, I performed multiple benchmarks on multiple data sets. All of the benchmarks were run on my own machine consisting of Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz (dual-core, 32K L1 cache, 256K L2 and 3072K L3 cache per core), 8 GB of RAM. The machine ran Ubuntu Linux (kernel version 18.04). Code was compiled on g++ version 7.2.0 with compiler optimizations enabled (g++ -O4). In the tables below, I display the best performing with color blue and the worst performing with color red.

2.1 Randomly generated sequences

2.1.1 Randomized inserts

Before testing fully randomized sequences, I tested inserting elements with random keys to each of the heaps I implemented, to see how they compare and how constant amortized insert times of all heaps except the binary heap translate to the reality. To have more generalized results, I tested sequences with different length, starting from 10^2 to 10^6, rising the power of 10 by 1. As for my expectations, I expected the heaps with constant amortized time, namely the binomial, the Fibonacci, the rank-pairing and the violation heap, to outperform the binary heap which should insert in $O(\log n)$ time. Normalized results of inserting elements is shown in table 2.1.

Table 2.1: Inserts (100 runs, normalized)

	10^2	10^3	10^4	10^5	10^6
binary (explicit)	3.65	2.44	2.45	2.60	2.72
binary (implicit)	3.71	2.49	2.18	1.89	1.78
binomial	11.19	10.61	11.04	10.92	11.39
Fibonacci	1.75	1.46	3.03	3.57	4.27
rank-pairing	1.00	1.00	1.00	1.00	1.00
violation	1.02	1.07	1.97	1.86	1.57

Reviewing the results, I find that relaxed versions of the Fibonacci heap, namely the rank-pairing and the violation heap, yield the expected results, being one of the fastest. The Fibonacci heap underperforms and starts to fall behind more as the number of inserts gets higher. Both variants of the binary heap, in comparison with the relaxed versions of the Fibonacci heap, perform better on larger sequences than on smaller ones, with the implicit version performing significantly better than the explicit. The binomial heap, being 20 times slower than the rank-pairing heap on 10^6 inserts, does not meet the constant amortized time and underperforms considerably. The biggest surprise is the binary heap outperforming the Fibonacci heap on 10^6 inserts as the expected difference in amortized run-times is not met.

2.1.2 Sorting

Next, I decided to test the sorting capabilities of each heap. Sorting translates to inserting a sequence of random key-item pairs and then repeatedly calling *ExtractMin* until the heap is empty. I performed this test on randomly generated sequences of insert operations with the length from 10^2 to 10^4 and then calling *ExtractMin* 10^2 - 10^4 times. Due to significant run-time increase with the larger sequences and limited computing power of my machine, I decided to not perform tests on sequences with insert count higher than 10^4. My expectations being shaken from the previous experiment, the binary heap should perform better than expected since the amortized time of *ExtractMin* is the same for every implemented heap.

However, as table 2.2 shows, it is clear that the implicit binary heap outperforms the other heaps on smaller sequences by small margin. The explicit binary heap starts slow but progressively speeds up and the difference between the implicit and the explicit version of binary heap gets less significant as the sequence grows in size. The binomial heap performs poorly on smaller sequences, but it rivals with the Fibonacci heap and its relaxed variants on larger ones. All of the relaxed versions of the Fibonacci heap perform worse than expected, as I expected them to be the fastest on larger sequences.

Table 2.2: Sorting (100 runs, normalized)

	10^2	10^3	10^4
binary (explicit)	1.23	1.04	1.11
binary (implicit)	1.00	1.00	1.00
binomial	3.55	2.05	2.12
Fibonacci	1.40	1.39	2.18
rank-pairing (t1)	1.31	1.38	2.29
rank-pairing (t2)	1.30	1.34	2.29
violation	1.41	1.40	2.30

2.1.3 Randomized inserts and decrease keys

DecreaseKey is proven to run in $O(1)$ amortized time in the Fibonacci-like heaps [3, 4, 5]. To test this, I generate uniformly distributed sequences of various lengths. These sequences contain only *Insert* and *DecreaseKey* operations and are not ordered in any way. With this setup, I wanted to accomplish different sizes of the inner structure of the heap variants so the experiment would yield more general results. However, the count of the executed operations will almost surely be less than the generated sequence, since *DecreaseKey* can be called on empty heaps. In this data set, I decided to allow such behavior.

Table 2.3: Insert followed by decrease key (100 runs, normalized)

	10^2	10^3	10^4	10^5
binary (explicit)	2.21	1.56	1.09	1.28
binary (implicit)	2.13	1.48	1.00	1.00
binomial	4.92	3.31	1.39	2.13
Fibonacci	1.33	1.14	1.35	1.71
rank-pairing (t1)	1.00	1.03	1.25	1.58
rank-pairing (t2)	1.05	1.00	1.30	1.55
violation	1.01	1.04	1.42	1.52

As seen in table 2.3, the constant amortized times give edge to Fibonacci-like heaps on smaller sequences, but they fall behind on larger ones. On smaller sequences, the performance of the binomial heap is the worst, but improves as the sequence length grows. On larger sequences, the binary heap variants reign supreme in terms of performance once again.

2.1.4 Randomized operations

Finally, I tested the heaps on randomly generated sequences with uniformly distributed operations of *Insert*, *DecreaseKey* and *ExtractMin*. To avoid possible exception throws that may occur, *ExtractMin* and *DecreaseKey* are ignored if a heap is empty and *Insert* is called instead. This decision ensures that the number of executed operations is the same and not less than the length of the generated sequence.

Table 2.4: Random operations (100 runs, normalized)

	10^2	10^3	10^4	10^5	10^6
binary (explicit)	1.13	1.31	1.24	1.16	1.13
binary (implicit)	1.00	1.00	1.00	1.00	1.00
binomial	2.47	4.20	3.50	2.57	1.62
Fibonacci	1.24	1.46	1.47	1.28	1.19
rank-pairing (t1)	1.31	1.56	1.54	1.31	1.20
rank-pairing (t2)	1.28	1.51	1.52	1.31	1.21
violation	1.23	1.52	1.54	1.34	1.20

Surprisingly, as seen in table 2.4, the binary heap once again outperformed every other heap implementation, with relaxed Fibonacci-like heaps being the closest. The explicit version of the binary heap repeatedly keeps pace with its implicit counterpart and gets better as the size of the sequence grows. The difference between type-1 and type-2 degree reduction in the rank-pairing heap gets less significant and both of these variants perform the same on large sequences.

2.1.5 Surprising results

Experimenting on artificially generated sequences showed that the binary heap performs surprisingly well compared with the other heaps with better amortized run-times. My take on this result is that more straight-forward implementations of the binary heap is optimized better by the compiler than the complicated implementations of other heap variants. To test this hypothesis, I turned the compiler optimizations off and I re-ran the benchmarking tests on sequences with randomized operations. Results are shown in table 2.5.

Table 2.5: Random operations (optimizations off, 100 runs, normalized)

	10^2	10^3	10^4	10^5	10^6
binary (explicit)	2.02	1.08	1.82	1.96	1.99
binary (implicit)	6.00	6.22	6.29	7.15	6.84
binomial	10.80	14.20	15.23	18.60	21.08
Fibonacci	1.27	1.29	1.58	1.80	1.99
rank-pairing (t1)	1.03	1.00	1.01	1.01	1.00
rank-pairing (t2)	1.00	1.00	1.00	1.00	1.00
violation	1.04	1.10	1.27	1.30	1.23

The impact of the compiler optimizations is apparent right from the start. Suddenly, the rank-pairing heap performs the best on both the smaller and the larger sequences. The implicit binary heap performs much worse with optimizations turned off, being the second worst heap in terms of performance. Both of the biggest underperformers, the binomial and the implicit binary heap, accumulate large differences in the run-time over the other heaps. What is shared only by the two is that I used STL [1] to implement these structures. As experiments show, the optimizations of STL containers have great impact on the overall performance of these heaps as the optimizations turned off made them unable to compete with the other heap variants.

1. https://en.cppreference.com/w/cpp/container

2.2 Graphs

As data sets for benchmarks on graphs, I decided to reuse graph data of Nathan Sturtevant's work [9]. These maps are stored in text files with .map extension and have the following format [9]:

- All maps begin with the lines:

  type octile
  height x
  width y
  map

  where x and y are the respective height and width of the map.

- The map data is store as an ASCII grid. The upper-left corner of the map is (0,0). The following characters are possible:

  . - passable terrain
  G - passable terrain
  @ - out of bounds
  O - out of bounds
  T - trees (unpassable)
  S - swamp (passable from regular terrain)
  W - water (traversable, but not passable from terrain)

From all of the possible terrains, I utilize only '.' as a passable terrain and '@' as an out of bounds indicator. Passable terrain translates to the vertex in the graph, with most vertices having 8 neighbors in mazes with larger corridor width and real-life maps. This results in dense graphs on which the Fibonacci heap and its relaxed variants should reign supreme. Distance-wise, all neighbors of a vertex v have distance of 1.

I divide the graph benchmarks into two categories: artificial generated mazes and real-life locations from Berlin, Boston, Denver, London, New York and Paris. All of the artificial maze maps are grids of 512 x 512 characters which are transformed into the graph with less than

262144 vertices. Real-life locations are larger, being grids of 1024 x 1024 characters. However, these maps have much larger obstacles than mazes, resulting in much smaller vertex count than 1048576.

Performance of the heap variants is tested by finding the shortest path from the first to the last vertex in the graph using two variants of Dijkstra's algorithm.

Dijkstra$_1$ is a traditional approach of Dijkstra's algorithm. All of the vertices in the graph are inserted to the heap, with the distance set to ∞ for each node except the source. *ExctractMin* is then called repeatedly, until the heap is empty or destination vertex is reached. In each iteration, *DecreaseKey* for each neighbor *n* of extracted minimum node is called, if the updated distance is less than the stored distance.

```
1  Dijktra1(source, destination):
2    h = new Heap()
3    for each vertex v in the graph:
4      if v == source:
5        v.dist = 0
6      else:
7        v.dist = ∞
8      insert v to h
9
10   while !h.Empty():
11     u = h.ExtractMin()
12     if u == destination:
13       return
14     for each neighbor v of u:
15       alt = u.dist + 1
16       if alt < v.dist:
17         v.prev = u
18         decrease key of v in h
19   error: destination was not found
```

Instead of filling the heap with all vertices, it is possible to insert only the source node. Then, inside the if statement starting on line 16, the vertex must be inserted if not already in the heap; otherwise *DecreaseKey* is called. I refer to this modification as *Dijkstra*$_2$.

2.2.1 Randomly generated mazes

As stated before, mazes are grids of 512 x 512 size. From many different mazes in Sturtevant's work, I picked the ones that differ the most, with the main difference being the corridor width. I refer to each maze by its respective name 512-x-0, where x is the corridor width in terms of vertices. A picture of each maze can be found in appendix A.

In the first test, I ran $Dijkstra_1$ on each of the mazes. I came to this test with the expectation of Fibonacci-like heaps performing the best, because *DecreaseKey* is invoked regularly, which should give an edge to the Fibonacci-like heaps with constant amortized time of *DecreaseKey*. As table 2.6 shows, the results are surprising.

Table 2.6: $Dijkstra_1$: maze512-x-0 (100 runs, normalized)

	32	16	8	4	2	1
binary (explicit)	2.78	2.52	2.31	2.29	2.26	2.21
binary (implicit)	1.00	1.00	1.00	1.00	1.00	1.00
binomial	11.38	10.37	8.68	9.85	9.92	10.02
Fibonacci	2.25	1.86	1.92	1.96	1.98	2.08
rank-pairing (t1)	2.59	2.02	2.02	2.21	2.01	2.38
rank-pairing (t2)	2.56	2.05	2.12	2.06	2.12	2.23
violation	2.40	2.06	2.13	2.13	2.07	2.13

Overall, all of the heaps except the implicit binary and the binomial heap perform similarly on simpler mazes with large corridors. On all mazes, by far the best performing heap is the implicit binary heap. Surprisingly, the Fibonacci heap performs better than its relaxed variants on every maze. The explicit binary heap outperforms the rank-pairing heap on the most complicated maze. The result that stands out the most is the terrible performance of the binomial heap. Even with the compiler optimizations enabled, the binomial heap failed to meet my expectations and performs on average 10 times worse than the implicit binary heap on every maze.

With the results of the previous benchmark in mind, my expectations changed. The implicit binary heap performing the best in $Dijkstra_1$, I expected this heap to perform well once again. On av-

erage, $Dijkstra_2$ utilizes less inserts than $Dijkstra_1$ and as experiments in subsection 2.1.1 showed, *Insert* is not the strongest point of binary heaps. This means that $Dijkstra_2$ favors the binary heaps more and that the other heap variants should be outperformed by the binary heaps by greater margin.

Table 2.7: $Dijkstra_2$: maze512-x-0 (100 runs, normalized)

	32	16	8	4	2	1
binary (explicit)	1.09	1.15	1.11	1.20	1.12	1.15
binary (implicit)	1.00	1.00	1.00	1.00	1.00	1.00
binomial	4.03	4.23	3.79	4.04	3.63	3.49
Fibonacci	1.51	1.65	1.50	1.68	1.52	1.61
rank-pairing (t1)	1.47	1.59	1.39	1.67	1.42	1.50
rank-pairing (t2)	1.46	1.62	1.35	1.62	1.46	1.47
violation	1.74	1.74	1.62	1.69	1.62	1.65

Reviewing the results in table 2.7, I find that this is exactly the case. The implicit binary heap is performing the best and its explicit counterpart is slightly slower. The binomial heap is the worst performer and the Fibonacci-like heaps are close to each other in terms of performance. The rank-pairing heaps managed to outperform the Fibonacci heap, as opposed to the benchmark of $Dijkstra_1$.

2.2.2 Real-life locations

Regarding the benchmarks of the real-life locations, I do not expect to see any significant changes in the results. I expect the implicit binary heap to be the fastest, with Fibonacci-like heaps close behind. To make the following tables more readable and compact, I am shortening the city names to only the first three characters of their respective names, with the exception of New York location being called NY. A picture of each location can be found in the appendix B.

Running $Dijkstra_1$ on every location yields the expected result, as shown in table 2.8. However, running $Dijkstra_2$ shows something strange. The explicit binary heap managed to outperform the implicit binary heap on almost every location. Comparing the results in tables

Table 2.8: $Dijkstra_1$: real-life locations (100 runs, normalized)

	Ber	Bos	Den	Lon	NY	Par
binary (explicit)	2.08	2.23	2.12	2.27	2.23	2.24
binary (implicit)	1.00	1.00	1.00	1.00	1.00	1.00
binomial	7.28	6.42	6.12	6.96	6.80	5.96
Fibonacci	1.34	1.40	1.51	1.38	1.40	1.31
rank-pairing (t1)	1.57	1.64	1.62	1.64	1.71	1.43
rank-pairing (t2)	1.49	1.68	1.62	1.67	1.61	1.44
violation	1.49	1.57	1.55	1.57	1.63	1.42

2.9 and 2.7, I find that the explicit binary heap is getting a visible performance boost, performing much better on real-life locations than on artificial mazes. This performance gain on real-life locations is visible on every heap variant, with the exception of the binomial heap performing roughly the same. It is thus safe to assume that the real-world data sets have positive impact on the performance.

Table 2.9: $Dijkstra_2$: real-life locations (100 runs, normalized)

	Ber	Bos	Den	Lon	NY	Par
binary (explicit)	1.00	1.02	1.00	1.00	1.00	1.00
binary (implicit)	1.00	1.00	1.01	1.03	1.00	1.02
binomial	3.53	3.77	3.40	3.31	3.32	3.48
Fibonacci	1.66	1.30	1.38	1.30	1.24	1.25
rank-pairing (t1)	1.23	1.26	1.17	1.21	1.18	1.14
rank-pairing (t2)	1.26	1.28	1.17	1.25	1.18	1.15
violation	1.42	1.52	1.36	1.37	1.38	1.61

3 Conclusion

The Fibonacci and Fibonacci-like heaps, namely the rank-pairing and the violation heap, have proven constant amortized run-time for *Insert* and *DecreaseKey* operations [3, 5, 4]. This should give these heaps an edge when they are compared with the binary and the binomial heap with worse logarithmic run-times [3, 6, 7]. Reviewing the results of all of the experiments I conducted, I find that this is not the case. The implicit binary heap performs better on every data set except randomized inserts. I have shown that the surprising results of the implicit binary heap are caused by the compiler optimizations. With these optimizations disabled, the implicit binary heap performed the second worst on sequences of randomized operations, as opposed to being the best performer with said optimizations enabled on the same data set. The impact of these optimizations is so big, that the implicit binary heap managed to outperform the Fibonacci heap and its relaxed variants on both mazes and real-life location graphs which should, in theory, heavily favor the Fibonacci, the rank-pairing and the violation heap. The binomial heap has more complex implementation than the binary heap while not really improving the amortized run-times, which shows in the results as the binomial heap performed the worst on almost every data set. Differences in the run-time between the rank-pairing and the violation heap are not significant on any data set. On mazes and real-life locations, the Fibonacci heap manages to perform better than its relaxed variants when *DecreaseKey* was invoked more regularly, but is outperformed by the rank-pairing heap when the number of *DecreaseKey* calls is lower. As experiments have shown, the nature of the data set has positive impact on the performance of the implemented heaps. All of the implemented heap variants perform better, with smaller run-time differences on the real-life data, compared to the artificial data.

A Mazes

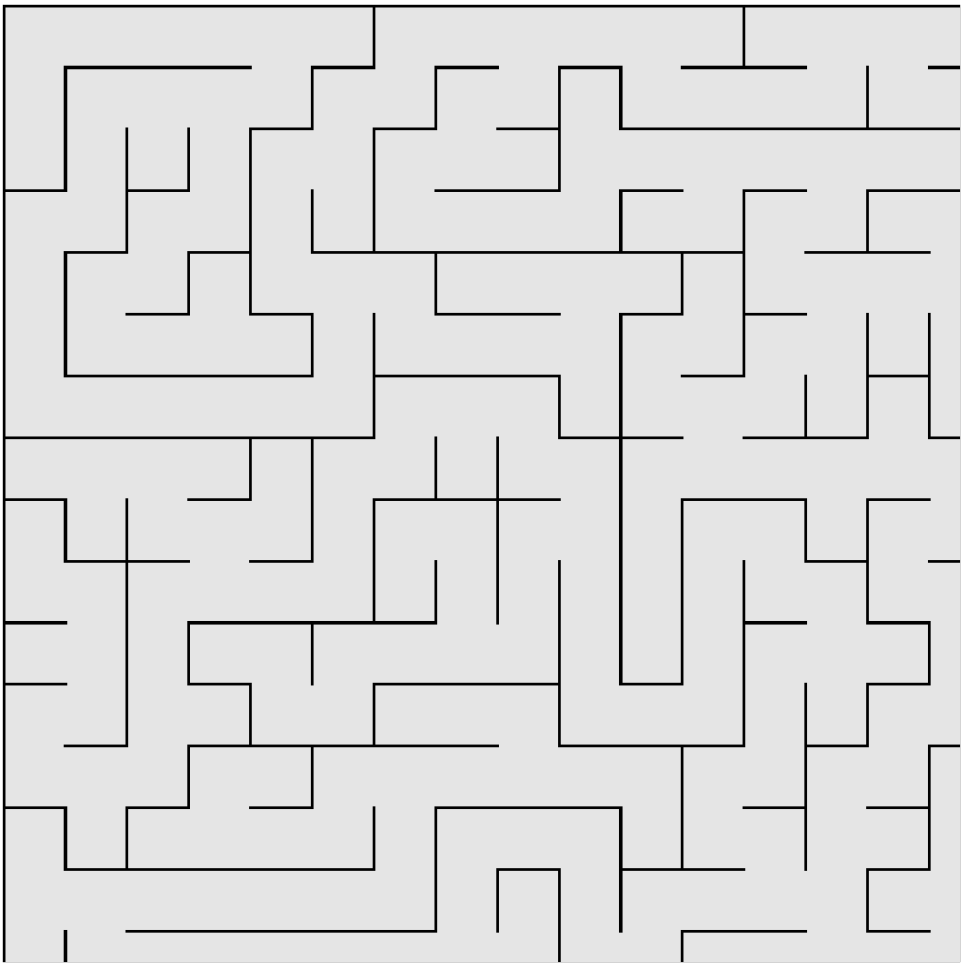

Figure A.1: maze 512-32-0

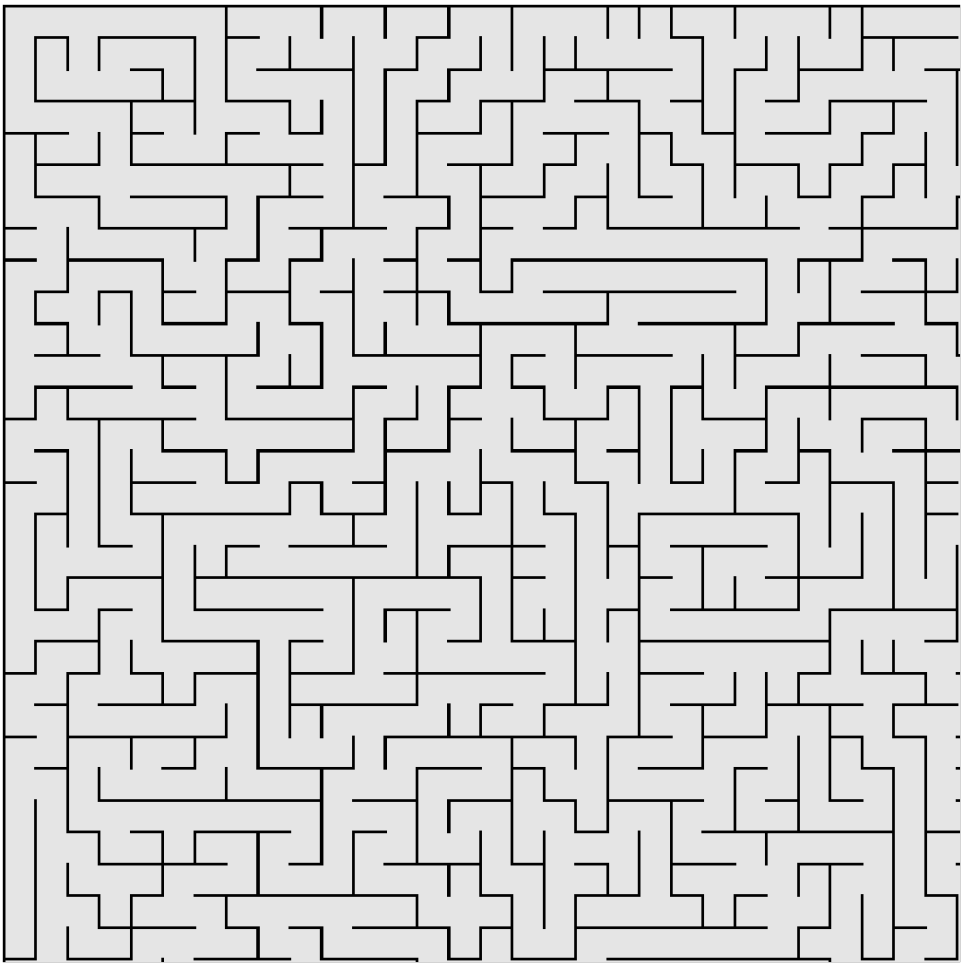

Figure A.2: maze 512-16-0

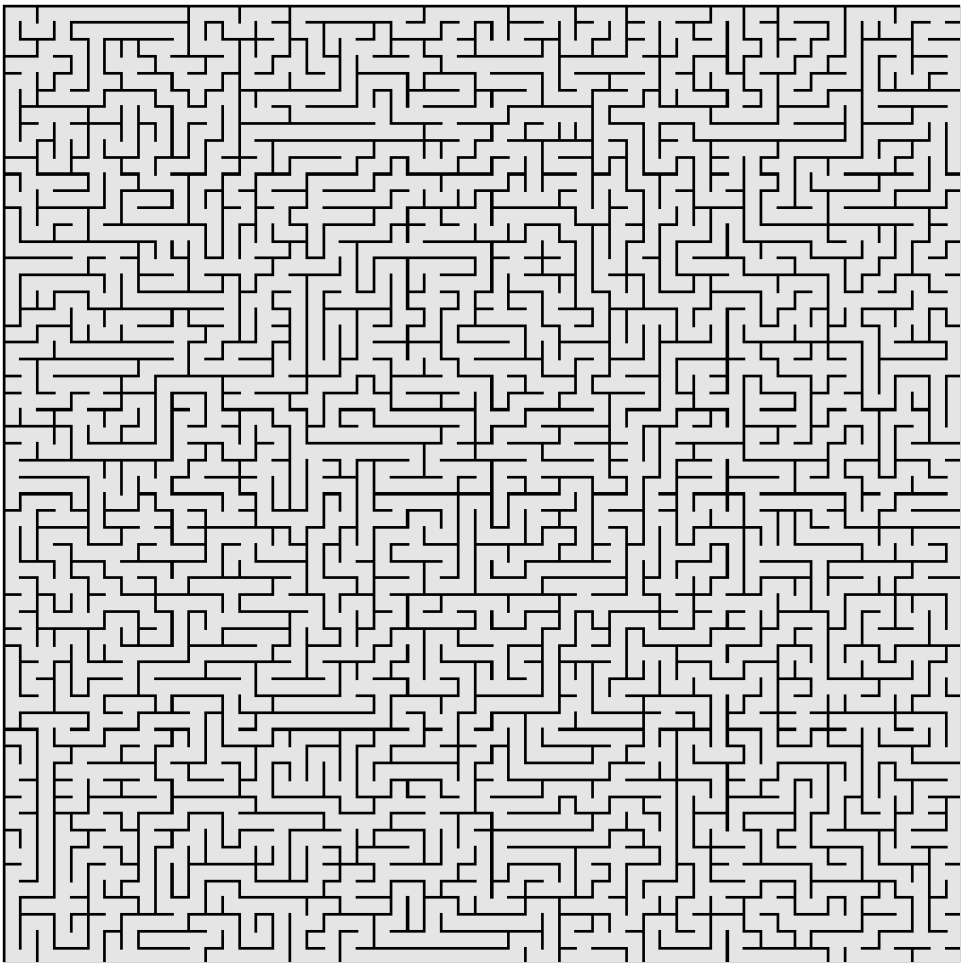

Figure A.3: maze 512-8-0

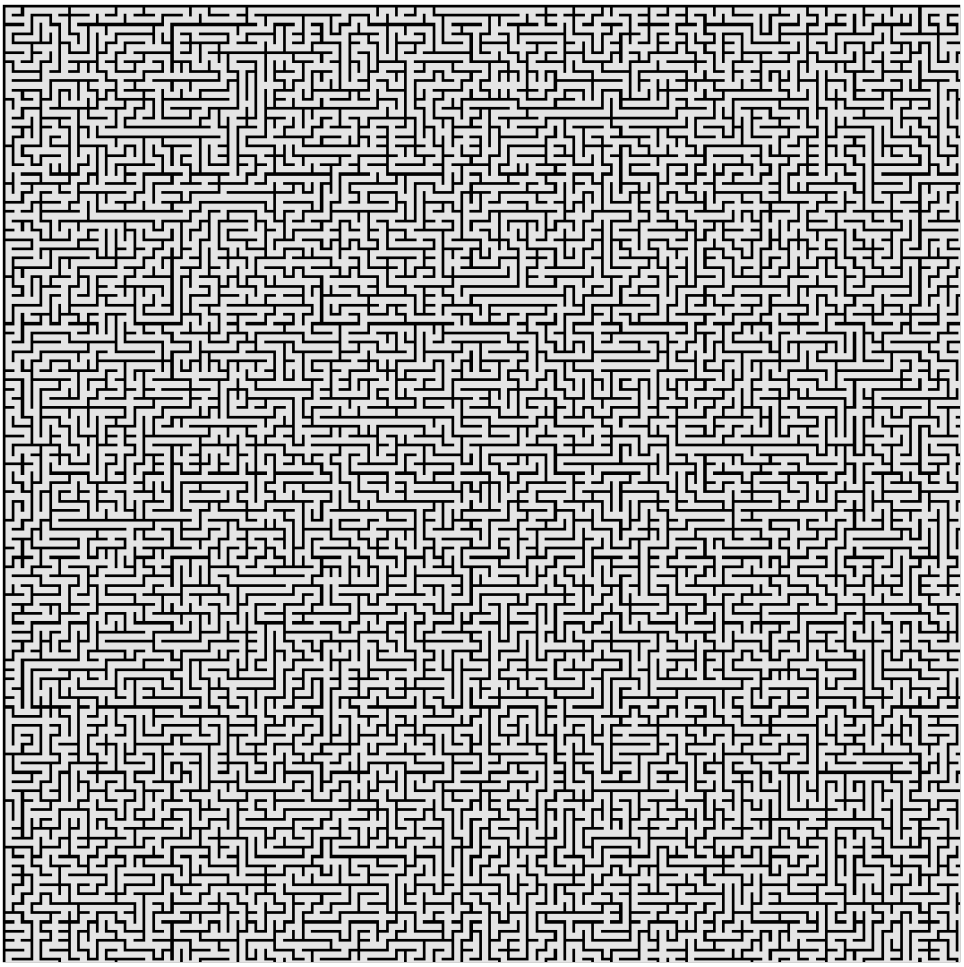

Figure A.4: maze 512-4-0

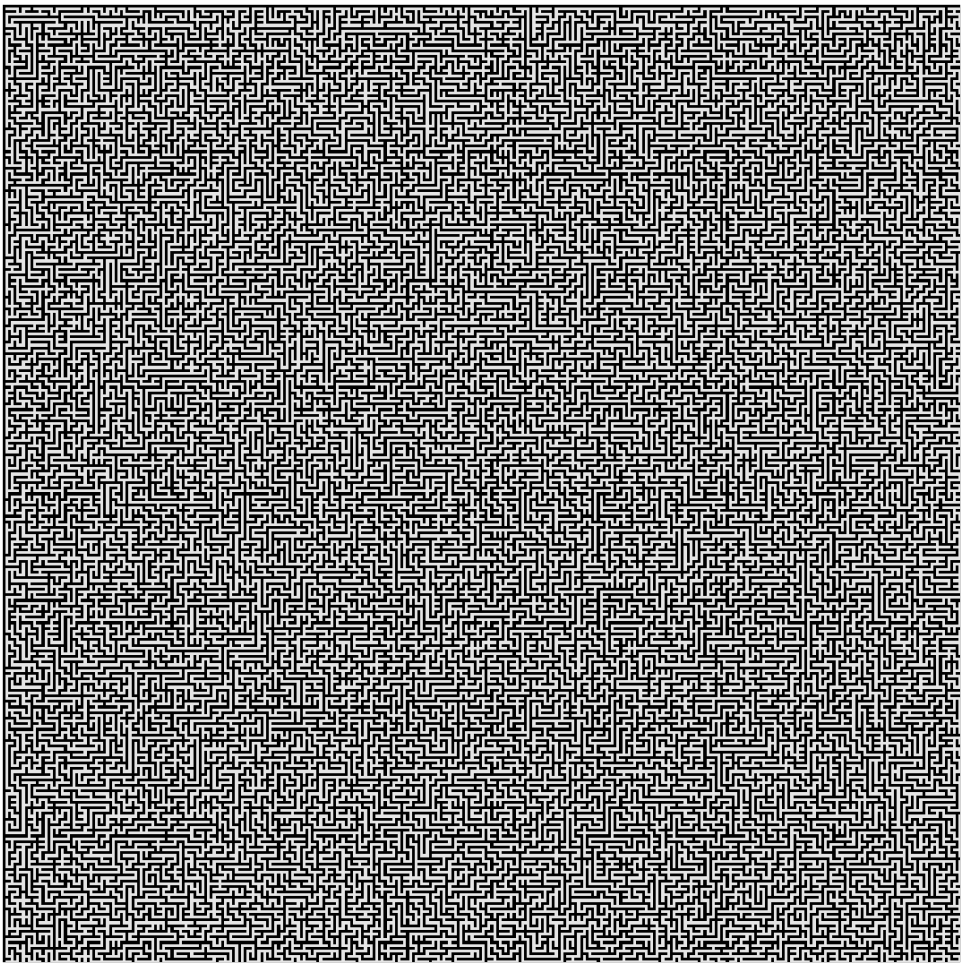

Figure A.5: maze 512-2-0

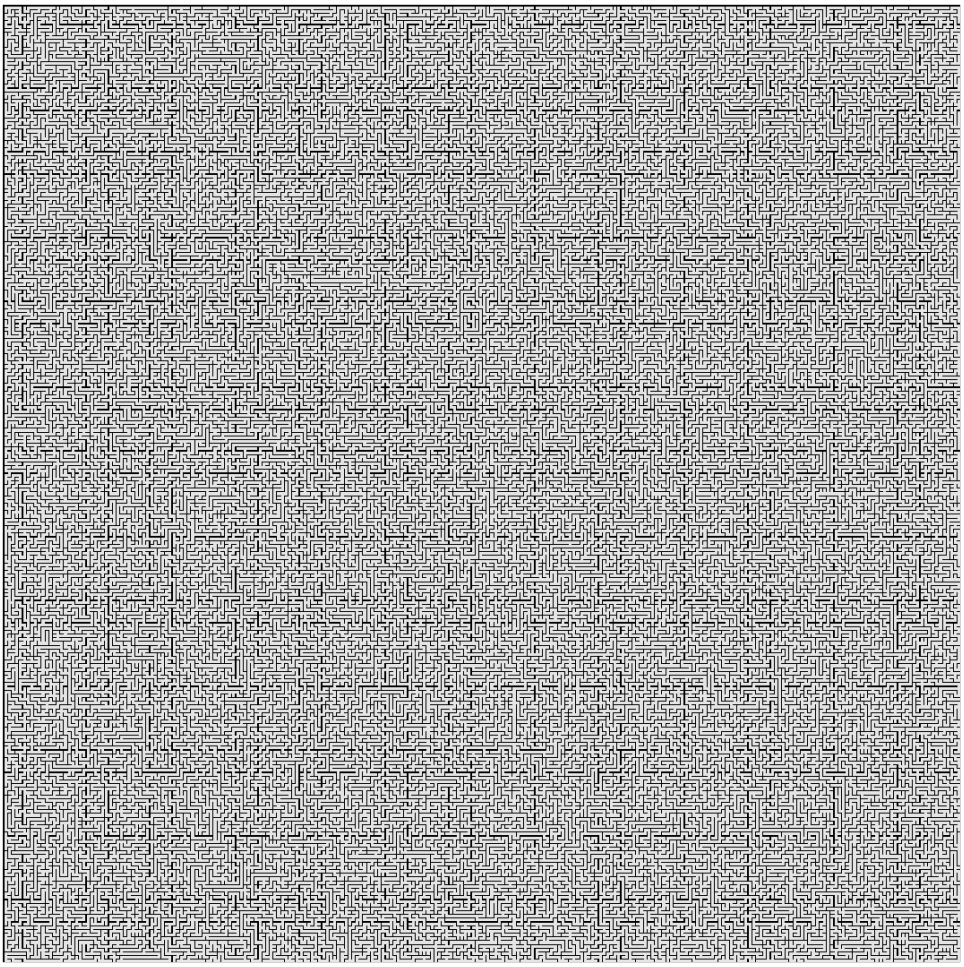

Figure A.6: maze512-1-0

B Real-life locations

Figure B.1: Berlin

Figure B.2: Boston

Figure B.3: Denver

Figure B.4: London

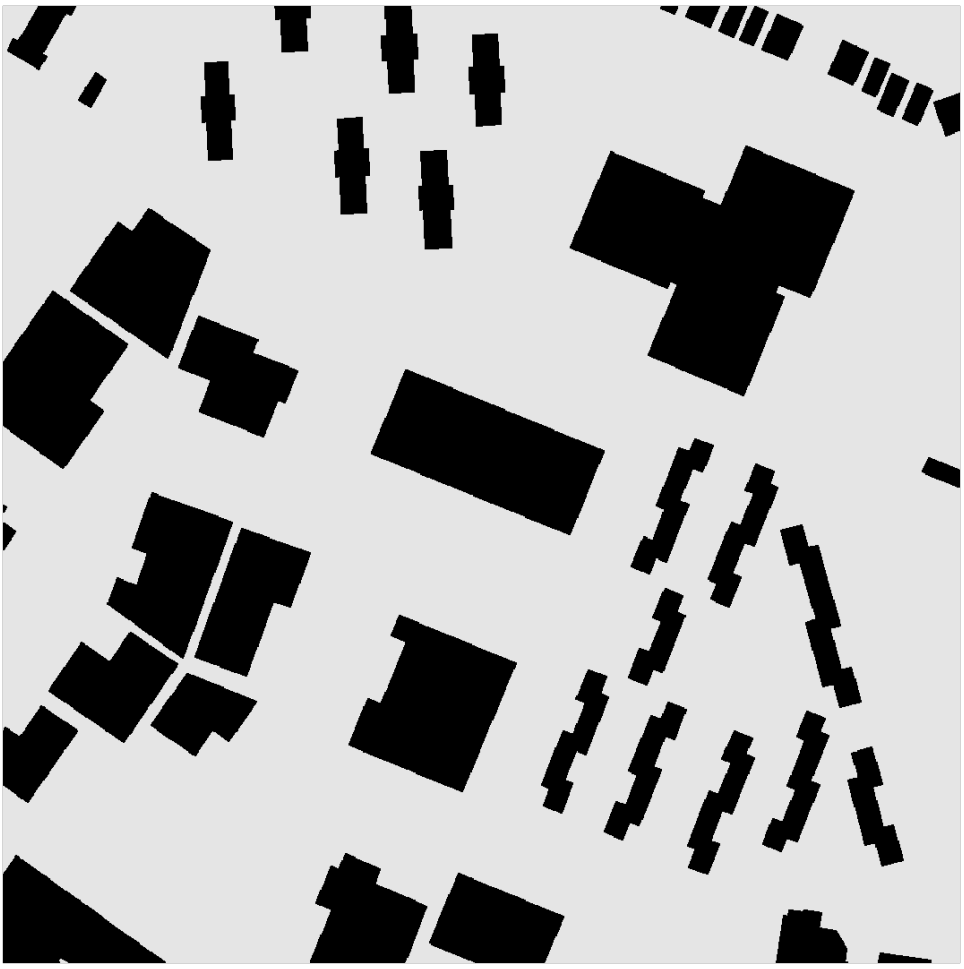

Figure B.5: New York

Figure B.6: Paris

Bibliography

1. KNUTH, Donald. *The Art of Computer Programming: Volume 3: Sorting and Searching*. 2nd ed. Addison Wesley Longman Publishing Co, 1998. ISBN 0-201-89685-0.
2. LARKIN, Daniel H.; SEN, Siddhartha; TARJAN, Robert Endre. A Back-to-Basics Empirical Study of Priority Queues. *CoRR*. 2014, vol. abs/1403.0252.
3. CORMEN, Thomas; LEISERSON, Charles; RIVEST, Ronald; STEIN, Clifford. *Introduction to Algorithms: Third Edition*. 3rd ed. The MIT Press, 2009.
4. HAEUPLER, Bernhard; SEN, Siddhartha; TARJAN, Robert E. Rank-Pairing Heaps. *SIAM J. Comput.* 2011, vol. 40, no. 6, pp. 1463–1485. ISSN 0097-5397.
5. ELMASRY, Amr. Violation Heaps: A Better Substitute for Fibonacci Heaps. *CoRR*. 2008, vol. abs/0812.2851.
6. *Is it possible to make efficient pointer-based binary heap implementations?* [online]. Ambroz Bizjak, 2016 [visited on 2018-11-18]. Available from: `https://stackoverflow.com/a/41338070`.
7. VUILLEMIN, Jean. A Data Structure for Manipulating Priority Queues. *Commun. ACM*. 1978, vol. 21, no. 4, pp. 309–315. ISSN 0001-0782. Available from DOI: `10.1145/359460.359478`.
8. FREDMAN, Michael L.; TARJAN, Robert Endre. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM*. 1987, vol. 34, no. 3, pp. 596–615. ISSN 0004-5411.
9. STURTEVANT, N. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*. 2012, vol. 4, no. 2, pp. 144–148. Available also from: `http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf`.