Radix heaps, an efficient implementation for priority queues

Jochen Könemann Christoph Schmitz
Christian Schwarz

MPI–I–95–1-009 March 1995

Radix heaps, an efficient implementation for priority queues[*]

Jochen Könemann[†] Christoph Schmitz[†] Christian Schwarz[‡]

March 14, 1995

Abstract

We describe the implementation of a data structure called radix heap, which is a priority queue with restricted functionality. Its restrictions are observed by Dijkstra's algorithm, which uses priority queues to solve the single source shortest path problem in graphs with nonnegative edge costs. For a graph with n nodes and m edges and real-valued edge costs, the best known theoretical bound for the algorithm is $O(m+n\log n)$. This bound is attained by using Fibonacci heaps to implement priority queues.

If the edge costs are integers in the range $[0\ldots C]$, then using our implementation of radix heaps for Dijkstra's algorithm leads to a running time of $O(m+n\log C)$. We compare our implementation of radix heaps with an existing implementation of Fibonacci heaps in the framework of Dijkstra's algorithm. Our experiments exhibit a tangible advantage for radix heaps over Fibonacci heaps and confirm the positive influence of small edge costs on the running time.

Contents

1. Introduction.

In this paper, we describe the implementation of a data structure named **radix heap**. For convenience, we will also refer to this structure as **r_heap** in the following.

A radix heap can be used to implement the data type *priority queue*. A priority queue stores a collection of items, each of which has a *key*. The keys in the queue come from a linearly ordered universe. There are operations available to update a priority queue. Operation *insert* adds a new item with a given key to the queue. Operation *decrease_key* takes a given item and decreases its key to a given value if applicable, i.e. if this value is smaller than the element's current key. Furthermore, with operation *del_item* we can delete any given item from the queue. The main query operation on a priority queue is *find_min*, which returns an item containing a key that is minimal among all keys occurring in the queue. Searching for any other particular key is not supported, and this is what sets priority queues apart from e.g. sorted sequences or dictionaries. For convenience, a

[*]This work was partially supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

[†]Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken and Computer Science Department, Universität des Saarlandes, D-66041 Saarbrücken

[‡]Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken

combination of the query *find_min* and the update *del_item* is regularly offered on priority queues as *del_min*.

Priority queues find an important application in the *shortest path problem:* given a graph G with n nodes and m edges where each edge is labeled with a nonnegative real-valued cost, the problem is to compute the shortest distance from a designated node s to all other nodes in G. A solution to this problem is provided by *Dijkstra's algorithm* [D59]. Its running time is dominated by n *insert*, *find_min* and *del_item* operations, plus m *decrease_key* operations carried out on a priority queue of size at most n. The theoretically most efficient algorithm is obtained by using *Fibonacci heaps* [FT87] to implement the priority queue. A Fibonacci heap provides *decrease_key* for $O(1)$ amortized time and the other mentioned operations for $O(\log n)$ amortized time. This leads to a running time of $O(m + n \log n)$ for Dijkstra's algorithm.

While this is optimal for real-valued edge costs, improvements are possible if edge costs are bounded integers. This was investigated by Ahuja et. al. [AMOT90]. Their paper contains two solutions and for both, the running time depends on the bound C given for the edge costs. While a one-level radix heap leads to a running time of $O(m + n \log C)$, a complicated two-level radix heap improves this bound to $O(m + n\sqrt{\log C})$. We chose to implement the one-level heap due to its simplicity in order to include it into the data structures library LEDA [MN89, Näh93, MN95].

Radix heaps exploit some special properties of Dijkstra's algorithm in the use of priority queues. For every node v of G, the algorithm maintains a tentative distance $d(v)$ that is not smaller than its actual distance to s, say $\widehat{d}(v)$. Each node $v \neq s$ has $d(v) = \infty$ at the start of the algorithm and enters the priority queue during the algorithm with a finite $d(v)$. It is stored there using $d(v)$ as its key until its distance to s is known, i.e. $d(v) = \widehat{d}(v)$. Assume that all edge costs are in the integer range $[0 \ldots C]$. Then the following properties hold:

(1) For any node v: if $d(v) < \infty$, then $0 \leq d(v) \leq nC$

(2) Let x be the node with minimal $d(x)$ in the priority queue. Then $d(x) = \widehat{d}(x)$. Furthermore, let v be any other node in the graph whose final distance $\widehat{d}(v)$ is not yet known. Then $\widehat{d}(v) \geq d(x)$, and if v is stored in the queued, we have $d(v) \in [d(x) \ldots d(x) + C]$.

Property (2) particularly means that the sequence of minimum keys is nondecreasing. A priority queue with the above properties is sometimes called a *bounded monotonic heap.* These restrictions have the following effect on the specification of the operations on **r_heaps**. Both *insert* and *decrease_key* are given a key value as one of their arguments. It is required that this key is not smaller than the current minimum key in the heap.

During the implementation, we will mention some parts of the analysis which are essential for the understanding of the implementation. The reader is referred to the original paper [AMOT90] for more details.

We implement the data structure **r_heap** in C++ with the intention to include it into the LEDA library. To achieve this goal, we follow LEDA conventions in the specification as well as in the implementation.

The paper is organized as follows. First, we give the specification of the data type in the header file `r_heap.h`. Then, we describe the implementation of the functions that were declared in the header file. Following this, we describe a LEDA implementation of Dijkstra's algorithm using generic priority queues. Using this algorithm, we compare the performance of radix heaps with Fibonacci heaps, the default implementation of priority queues provided by LEDA. We conclude the paper with experimental results.

2. The header file.

The header defines two classes, one to represent a single heap element, and another to describe the structure of the heap itself, along with the operations defined on the heap. In the description of both classes, we use **GenPtr**, the generic pointer type of LEDA. Substituting actual types for an application will be done by LEDA's priority queue interface, with which we integrate our implementation. See [MN95] for more details on this issue.

Let us begin with some terminology. In a radix heap, elements are grouped into *buckets* according to their key. The first bucket will always contain the elements with the current minimum key (and only those), which enables us to carry out *find_min* efficiently.

The maximal difference between any two keys in a radix heap is a certain integer C which is given when the heap is created. There are $B = \lceil\log(C+1)\rceil + 2$ buckets. For each bucket $i, 0 \le i < B$, there is a number $u[i]$ which gives an upper bound on the keys stored in that bucket. More specifically, let min be the minimum key stored in the heap, and define $u[-1] = min - 1$. Then, as mentioned above, $u[0] = min$, and

1. $u[-1] < u[0] \le u[1] \le \ldots \le u[B-1]$,
2. an element with key k belongs to bucket i, where $0 \le i \le B-1$, if $u[i-1] < k \le u[i]$,
3. $u[i] \le u[i-1] + 2^{i-1}$ for $1 \le i < B-1$,
4. $u[B-1] = \text{MAXINT}$.

3. A single heap element is stored in an **r_heap_node**. First of all, this structure contains a *key* field. The field *inf* holds associated information. The *bucket* field gives the number of the bucket which currently contains the element. We want to maintain the elements of a bucket in a doubly linked list. Pointers *succ* and *pred* are provided for this purpose. For convenience, we also define **r_heap_item** as a pointer to an **r_heap_node**. Activation of the LEDA memory manager will improve the efficiency of memory (de-)allocation with **new** and **delete**.

```
⟨ r_heap_node 3 ⟩ ≡
  class r_heap_node {
    friend class r_heap;
    GenPtr key;      // key
    GenPtr inf;      // information
    int bucket;    // number of bucket containing the node
    r_heap_node *succ, *pred;     // pointers for list maintenance
  public:
    r_heap_node(GenPtr k, GenPtr i): key(k), inf(i), bucket(0), succ(nil), pred(nil) { }
    LEDA_MEMORY(r_heap_node)
  };
  typedef r_heap_node *r_heap_item;
```
This code is used in websection 6.

4. Class **r_heap** defines the radix heap itself. First, there is a constant C for the maximal key span in the heap. That is, the keys stored in the heap always come from an integer range $[min \ldots min + C]$. Each bucket maintains a list of **r_heap_node**s contained in it. Access to the buckets is provided by the array *buckets*[ ]. Along with this array, there is another array $u[\,]$ such that for any i, the keys of the elements stored in *buckets*$[i]$ are bounded by $u[i]$. Both arrays change dynamically during the lifetime of an **r_heap**. We will need to adjust the bucket boundaries recorded in $u[\,]$ from time to time, and to do this more efficiently, we tabulate the appropriate key range for the buckets in an array *bsize*[]. This array remains unchanged during the lifetime of an **r_heap**. B denotes the number of buckets necessary to store the elements in the heap for a given key span C.

The class declaration provides all operations that are contained in the LEDA priority queue interface, because we want to use **r_heap** in this framework. The counter *si* records the number of elements currently stored in the heap. Its purpose is to make the operations *empty* and *size* particularly simple and efficient.

```
⟨ class r_heap 4 ⟩ ≡
  class r_heap {
    /* data kept in an r_heap */
    int C;     // maximal difference between two keys in the heap
    r_heap_item *buckets;     // buckets of the r_heap
    int *u;     // upper bounds on the key intervals corresponding to the buckets
    int B;     // number of buckets
    int si;     // current number of elements stored in the heap
    int *bsize;     // table used to (re-)initialize the array u or part of it
    /* private functions that facilitate the descriptions of the r_heap operations */
    inline void set_bucket_bounds(int min, int upto);
    inline int findbucket(r_heap_item, int);
    void copy_heap(const r_heap &);
    inline void add_item(r_heap_item, int);
    inline void rm_item(r_heap_item);
    /* non-public functions concerned with the use of r_heap within LEDA */
    virtual void print_key(GenPtr) const { }
    virtual void print_inf(GenPtr) const { }
    virtual void clear_key(GenPtr &) const { }
    virtual void clear_inf(GenPtr &) const { }
    virtual void copy_key(GenPtr &) const { }
    virtual void copy_inf(GenPtr &) const { }
    virtual int int_type() const { return 0; }
  protected:
    r_heap_node *item(void *p) const { return (r_heap_node *) p; }
  public:
    r_heap(int C);
      // the maximal difference between two keys in the heap needs to be provided upon initialization
    r_heap() { error_handler(1, "r_heap:␣must␣initialize␣with␣int␣C>=0"); }
    r_heap(const r_heap &);
    r_heap &operator=(const r_heap &);
    virtual ~r_heap() { clear(); }
    r_heap_item find_min() const;
    r_heap_item insert(GenPtr k, GenPtr i);     // precondition: k ≥ key(find_min())
    void del_item(r_heap_node *x);
    void del_min();
    void decrease_key(r_heap_node *x, GenPtr k);
      // precond.: key(find_min()) ≤ k < key(x)
    void change_inf(r_heap_node *x, GenPtr i);
    GenPtr key(r_heap_node *x) const { return x→key; }
    GenPtr inf(r_heap_node *x) const { return x->inf; }
    void clear();
    int size() const;
    int empty() const;
    /* functions that are used by the LEDA iteration macros */
    r_heap_item first_item() const;
    r_heap_item next_item(r_heap_item p) const;
    void print_contents(ostream &chk = cout) const;
  };
  /* dummy I/O and cmp functions */
  inline void Print(const r_heap &, ostream &) { }
  inline void Read(r_heap &, istream &) { }
  inline int compare(const r_heap &, const r_heap &) { return 0; }
```
This code is used in websection 6.

5. It is necessary to include a header file containing standard functionality of LEDA.

```
⟨ includes 5 ⟩ ≡
#include <LEDA/basic.h>
```
This code is used in websection 6.

6. The header file now looks as follows:

```
⟨ r_heap.h  6 ⟩ ≡
  ⟨ includes 5 ⟩
  ⟨ r_heap_node 3 ⟩
  ⟨ class r_heap 4 ⟩
```

7. The implementation.

In the following sections, we present an annotated implementation of the member functions of class **r_heap**.

8. We need a constructor that creates an empty **r_heap**. In the previous section, we have seen that class **r_heap** does not permit a constructor without arguments. Instead, there is a constructor that receives an **int** argument. This argument determines the constant C and is used to calculate B, the number of buckets in the heap. The constructor also allocates space for the arrays *buckets*[ ], *u*[ ] and *bsize*[ ]. After this, *buckets*[ ] and *bsize*[ ] are initialized. The array *u*[ ] is supposed to keep the upper bounds on the keys stored in the buckets. These values cover the range $[min \ldots min + C]$ of keys stored in the heap, where *min* is the current minimum key. Since the invocation of this constructor creates an empty heap, we do not know *min* and defer the initialization of array *u*[ ] until the first *insert* operation. Nevertheless, we already set the last entry, $u[B-1]$, whose value MAXINT remains unchanged throughout the lifetime of the **r_heap**.

⟨ constructors 8 ⟩ ≡

```
  r_heap :: r_heap(int c)
  {
    C = c;
    B = int(ceil(log(C)/log(2))) + 2;
    buckets = (r_heap_item *) new int [B];
    for (int i = 0; i < B; i++) buckets[i] = nil;
    bsize = new int [B];
    u = new int [B];
    bsize[0] = 1;
    bsize[B − 1] = MAXINT;
    for (i = 1; i < B − 1; i++) bsize[i] = 1 ≪ (i − 1);
    u[B − 1] = MAXINT;
    /* this value won't change throughout the computation the other u[] values will be initialized
    by insert */
    si = 0;
  }
```

See also websection 11.

This code is used in websection 28.

9. Throughout this section we need functions that add an **r_heap_item** to—or remove it from—a bucket. Each of the following functions operates on a doubly linked list which connects the **r_heap_items** that belong to a bucket.

⟨ private functions 9 ⟩ ≡

```
  inline void r_heap :: add_item(r_heap_item it, int bnr)
  {
    it→succ = buckets[bnr];
    if (buckets[bnr] ≠ nil) buckets[bnr]→pred = it;
    it→pred = nil;
    it→bucket = bnr;
    buckets[bnr] = it;
  }
  inline void r_heap :: rm_item(r_heap_item it)
  {
    if (it→pred ≠ nil) (it→pred)→succ = it→succ;
    else buckets[it→bucket] = it→succ;
    if (it→succ ≠ nil) (it→succ)→pred = it→pred;
  }
```

See also websections 10, 12, and 16.
This code is used in websection 28.

10. For the copy constructor and for the similar assignment operator, which are described below, we use an auxiliary function. Called for an **r_heap** object, the function *copy_heap* is given another **r_heap** as its argument and replicates that heap in its own member variables.

⟨ private functions 9 ⟩ +≡

```
  void r_heap :: copy_heap(const r_heap &rh)
  {
    C = rh.C;
    B = rh.B;
    si = rh.si;
    buckets = (r_heap_item *) new int [B];
    u = new int [B];
    bsize = new int [B];
    for (int i = 0; i < B; i++) {
      u[i] = rh.u[i];
      bsize[i] = rh.bsize[i];
    }
    r_heap_item item1, item2;
    for (i = 0; i < rh.B; i++) {
      if (rh.buckets[i] ≠ nil) {
        item1 = rh.buckets[i];
        do {
          item2 = new r_heap_node (item1→key, item1→inf);
          add_item(item2, i);
          item1 = item1→succ;
        } while (item1 ≠ nil);
      }
      else buckets[i] = nil;
    }
  }
```

11. Using the function *copy_heap*, the copy constructor and the assignment operator are simple.

⟨ constructors 8 ⟩ +≡

```
  r_heap :: r_heap(const r_heap &rh)
  {
    copy_heap(rh);
  }
  r_heap &r_heap :: operator=(const r_heap &rh)
  {
    if (this ≠ &rh) {
      delete [ ]buckets;
      delete [ ]u;
      copy_heap(rh);
    }
  }
```

12. During the lifetime of an **r_heap**, we often change the bucket boundaries. The *set_bucket_bounds* function serves this purpose. Its arguments are a key value *min* and a bucket number *upto*. The function sets $u[0] = min$ and then redefines $u[1], \ldots, u[upto - 1]$. Since it is a private function, we do not check these arguments for integrity, but the requirements are $1 < upto < B - 1$ and $u[0] \leq min \leq u[upto]$.

According to the original paper [AMOT90], we need compute $u[i] = \mathrm{Min}(u[i-1]+bsize[i], u[upto])$ for $i = 1, \ldots, upto - 1$. However, we know that the sequences $u[]$ and $bsize[]$ are monotonically nondecreasing, so the outcome of the above Min computation is also monotone. Hence, when computing $u[i]$, we explicitly check whether $u[i-1] + bsize[i] > u[upto]$. If this is the case, then $u[j] = u[upto]$, $i \leq j \leq upto - 1$, which simplifies the computation for these j.

```
⟨ private functions 9 ⟩ +≡
  inline void r_heap :: set_bucket_bounds(int min, int upto)
  {
    u[0] = min;
    for (int i = 1; i < upto; i++) {
      u[i] = u[i − 1] + bsize[i];
      if (u[i] > u[upto]) break;
    }
    for ( ; i < upto; i++) u[i] = u[upto];
  }
```

13. We are now ready to describe the implementation of the important priority queue operations. Our implementation slightly deviates from the one sketched in [AMOT90] by additionally maintaining the following invariant for a non-empty **r_heap**:

> Before and after each operations, the elements with minimum key are contained in the first bucket of the **r_heap**, i.e. in *bucket*[0].

14. Operation *find_min*, which returns an element with minimum key, has a simple implementation due to the aforementioned invariant.

```
⟨ public functions 14 ⟩ ≡
  r_heap_item r_heap :: find_min(void) const
  {
    if (si > 0) return buckets[0];
    else error_handler(1, "r_heap::find_min␣:␣Heap␣is␣empty!");
  }
```

See also websections 15, 17, 18, 19, 20, 21, 24, 25, and 26.
This code is used in websection 28.

15. Operation *insert* adds a new element the heap. To do this, it is given two arguments which represent a key and an information, respectively. First, an **r_heap_node** is created using the key and the information argument. If the heap was previously empty, we set the interval bounds using the key argument and put the newly created **r_heap_node** into the first bucket. Otherwise, the bucket bounds are already initialized, and to find the correct bucket for the new element, we scan the bucket boundaries for the "slot" containing the new key, in descending order starting with the last bucket.

After the new **r_heap_node** is inserted into the data structure in either way, we conclude the operation by updating the element counter and returning a pointer to the new node.

```
⟨ public functions 14 ⟩ +≡
  r_heap_item r_heap :: insert(GenPtr k, GenPtr i)
  {
    r_heap_item item = new r_heap_node (k, i);
    if (si > 0) {
      /* We check whether the operation respects the r_heap conditions */
      if (int(k) < u[0] ∨ int(k) > u[0] + C) {
        string s("r_heap::insert:␣k=␣%d␣out␣of␣range␣[%d,%d]\n", int(k), u[0], u[0] + C);
        error_handler(1, s);
      }
      int i = findbucket(item, B − 1);
      add_item(item, i);
    }
    else {
      set_bucket_bounds((int) k, B − 1);
      buckets[0] = item;
      item→bucket = 0;
    }
    si++;
    return item;
  }
```

16. We haven't described *findbucket* yet. This is a private function which, given an **r_heap_item** as its first argument, finds the appropriate bucket for that item. It does this by scanning the buckets in decreasing order starting from the bucket whose number is given as the second argument. Since *find_bucket* is private, we do not check any invariants. This is up to the public functions calling *find_bucket*.

```
⟨ private functions 9 ⟩ +≡
  inline int r_heap :: findbucket(r_heap_item it, int start)
  {
    if (int(it→key) ≡ u[0]) start = −1;
    else
      while (int(it→key) ≤ u[−−start]) ;      // now u[start] < int(it− > key) ≤ u[start + 1]
    return (start + 1);
  }
```

17. Operation *del_min* may be expressed using *find_min* and *del_item*. For efficiency reasons, we replace the call of *find_min* by a direct access to the head of the first bucket, since our invariant guarantees that *bucket*[0] is not empty if the heap is not empty.

```
⟨ public functions 14 ⟩ +≡
  void r_heap :: del_min(void)
  {
    if (si > 0) {
      r_heap_item it = buckets[0];
      del_item(it);
    }
    else error_handler(1, "r_heap::del_min␣:␣Heap␣is␣empty!");
  }
```

18. Operation *decrease_key* is given an item x sets its key to a given value k. According to the specification given before, the new value must be smaller than the previous key of x but not smaller than the current minimum key of the heap. If these requirements are satisfied, the key of the item is decreased as desired. Additionally, if the new key violates the boundaries of the item's bucket, the item is moved to a bucket with lower index.

⟨ public functions 14 ⟩ +≡
```
void r_heap :: decrease_key(r_heap_node *x, GenPtr k)
{
  if ((int(k) < int(x→key)) ∧ (int(k) ≥ u[0])) {
    x→key = k;
    if ((int(k) ≤ u[x→bucket − 1])) {
      rm_item(x);
      int i = findbucket(x, x→bucket);
      add_item(x, i);
    }
  }
  else {
    string s("r_heap::decrease_key:␣k=␣%d␣out␣of␣range␣[%d,%d]\n", int(k), u[0],
        int(x→key) − 1);
    error_handler(1, s);
  }
}
```

19. The following operation allows the user to change the associated information of an item. It has no influence on the heap structure, however, since we can directly access the information by the given **r_heap_item**.

⟨ public functions 14 ⟩ +≡
```
void r_heap :: change_inf(r_heap_node *x, GenPtr i)
{
  x→inf = i;
}
```

20. Operation *clear* deletes all elements of the **r_heap** and deallocates the storage space taken by the elements.

⟨ public functions 14 ⟩ +≡
```
void r_heap :: clear(void)
{
  r_heap_item it;
  for (int i = 1; i < B; i++)
    while ((it = buckets[i]) ≠ nil) {
      rm_item(it);
      delete it;
    }
}
```

21. Operation *del_item* removes a given item from the heap. This is the most complicated of our operations. If the heap is not empty after the given item has been removed, but the first bucket does no longer contain any element, then the invariant is violated. To reinstate the invariant, we first find the element that holds the minimum key. We place this element in *bucket*[0] and readjust the bucket boundaries accordingly. Finally, all the elements which previously were in the same bucket as the new heap minimum are redistributed to buckets with lower indices according to the new boundaries.

```
⟨ public functions 14 ⟩ +≡
  void r_heap :: del_item(r_heap_node *x)
  {
    int buck = x→bucket;
    rm_item(x);
    delete x;
    if ((si > 1) ∧ (buck ≡ 0) ∧ (buckets[0] ≡ nil)) {
      r_heap_item item;
      ⟨ find new minimum 22 ⟩⟨ reorganize r_heap 23 ⟩
    }
    si --;
  }
```

22. We now describe how to find the element that holds the new minimum key. Knowing that *bucket*[0] is empty, we check the buckets in ascending order starting from *bucket*[1] until we find the first non-empty bucket. Note that such a bucket must exist since the case that the heap is empty after the operation was excluded by the previous **if** condition. We check all items of the found bucket to retrieve the one with the smallest key.

```
⟨ find new minimum 22 ⟩ ≡
  int idx = 1;
  while (buckets[idx] ≡ nil) idx++;
  item = buckets[idx];
  r_heap_item dummy = item→succ;
  while (dummy ≠ nil) {
    if ((int) dummy→key < (int) item→key) item = dummy;
    dummy = dummy→succ;
  }    // we have found the minimum
```

This code is used in websection 21.

23. We now reorganize the heap. The element that was found by the previous code segment will be the new minimum of the heap. Then we recalculate the new bucket boundaries for all buckets with index $1 < i < idx$. Then the new minimal element is moved to *bucket*[0], and for all the other elements in *bucket*[*idx*], we scan the new bucket boundaries starting from $u[idx - 1]$ in descending order and move the element to the appropriate bucket.

It is worth noting that out of the newly computed bucket boundaries, at least the last one, $u[idx - 1]$, equals $u[idx]$. It follows that *every* element of *bucket*[*idx*] must move to a bucket with smaller index. This allows us to amortize the time previously spent to find the new minimum.

```
⟨ reorganize r_heap 23 ⟩ ≡
  set_bucket_bounds(int(item→key), idx);
  rm_item(item);
  add_item(item, 0);
  /* Redistribution */
  item = buckets[idx];
  r_heap_item next;
  while (item ≠ nil) {
    next = item→succ;
    /* we know that every item in bucket #idx MUST be redistributed */
    rm_item(item);
    int i = findbucket(item, idx);
    add_item(item, i);
    item = next;
  }
```
This code is used in websection 21.

24. The operations *size* and *empty* can be easily implemented using the counter *si*.

```
⟨ public functions 14 ⟩ +≡
  int r_heap :: size(void) const
  {
    return si;
  }
  int r_heap :: empty(void) const
  {
    return (si ≡ 0);
  }
```

25. The following functions are used for iterating over the heap elements. There is a LEDA macro **forall_items** which is based on these functions. The function *first_item* is particularly simple because of our heap invariant. The function *next_item* checks whether the given item has a successor in its bucket. If this is the case, that successor is returned. Otherwise, the following buckets are checked until a non-empty bucket is found. If the search is not successful, the given item is the last one and *next_item* returns *nil*. Otherwise, the first element of the found bucket is returned.

```
⟨ public functions 14 ⟩ +≡
  r_heap_item r_heap :: first_item(void) const
  {
    return buckets[0];      // nil if heap is empty
  }
  r_heap_item r_heap :: next_item(r_heap_item p) const
  {
    if (p→succ ≠ nil) return p→succ;
    else {
      int next = p→bucket + 1;
      while ((next < B) ∧ (buckets[next] ≡ nil)) next++;
      if (next ≡ B) return nil;
      else return buckets[next];
    }
  }
```

26. The following function gives a overview of the contents of an **r_heap**. It is added for maintenance purposes.

```
⟨ public functions 14 ⟩ +≡
  void r_heap :: print_contents(ostream &os) const
  {
    r_heap_item item;
    os ≪ "--------------------------------------\n";
    os ≪ "Si:␣" ≪ si ≪ "\n";
    os ≪ "--------------------------------------\n";
    for (int i = 0; i < B; i++) {
      os ≪ "--------------------------------------\n";
      os ≪ "Bucket␣" ≪ i ≪ "\n";
      os ≪ "Intervall:␣[";
      if (i > 0) os ≪ u[i − 1] − 1;
      else os ≪ u[i];
      os ≪ "," ≪ u[i] ≪ "]\n";
      os ≪ "--------------------------------------\n";
      item = buckets[i];
      while (item ≠ nil) {
        os ≪ "Key:␣" ≪ (int) item→key ≪ "␣Bucket:␣" ≪ item→bucket;
        os ≪ "\n";
        item = item→succ;
      }
    }
  }
```

27. Here are the header files that need to be included for the implementation of **r_heap**.

```
⟨ r_heap includes 27 ⟩ ≡
#include "r_heap.h"
#include <math.h>
```

This code is used in websection 28.

28. The source code of the **r_heap** implementation is composed of the following chunks.

```
⟨ r_heap.c 28 ⟩ ≡
  ⟨ r_heap includes 27 ⟩
  ⟨ constructors 8 ⟩
  ⟨ private functions 9 ⟩
  ⟨ public functions 14 ⟩
```

29. Dijkstra's algorithm.

Here is the LEDA implementation of Dijkstra's algorithm. It uses the generic LEDA type **p_queue**. We will use this algorithm to evaluate the performance of our **r_heap** implementation. Note that the function *decrease_key* described in the definition of **r_heap** is replaced by *decrease_p*. The reason for this is a LEDA naming convention. Our implementation is integrated into the generic priority queue framework by an interface thats uses **r_heap** as an *implementation parameter* for **p_queue** (see also the next section). While this interface e.g. offers an operation *decrease_p*, it actually expects the implementation to provide the corresponding function with name *decrease_key* and explicitly performs the switch. More information on this issue can be found in the textbook on LEDA [MN95].

```
⟨ function dijkstra 29 ⟩ ≡
  void dijkstra(graph &G, node s, edge_array⟨int⟩ &cost, node_array⟨int⟩
          &dist, node_array⟨edge⟩ &pred, p_queue⟨int, node⟩ &PQ)
  {
    node_array⟨pq_item⟩ I(G);
    node v;
    forall_nodes (v, G) {
      pred[v] = nil;
      dist[v] = MAXINT;
    }
    dist[s] = 0;
    I[s] = PQ.insert(0, s);
    while (¬PQ.empty( )) {
      pq_item it = PQ.find_min( );
      node u = PQ.inf(it);
      int du = dist[u];
      edge e;
      forall_adj_edges (e, u) {
        v = G.target(e);
        int c = du + cost[e];
        if (c < dist[v]) {
          if (dist[v] ≡ MAXINT) I[v] = PQ.insert(c, v);
          else PQ.decrease_p(I[v], c);
          dist[v] = c;
          pred[v] = e;
        }
      }
      PQ.del_item(it);
    }
  }
```

This code is used in websection 30.

30. The benchmark program.

In this section, we show a simple program that compares radix heaps with Fibonacci heaps, LEDA's default implementation for priority queues. The declaration **p_queue**⟨**int**, **node**⟩ creates a Fibonacci heap whose priorities are integers, each of which has a **node** associated with it. An **r_heap** with maximal key span M is provided as a priority queue with the declaration **p_queue**⟨**int**, **node**, **r_heap**⟩ (M). With this mechanism, **r_heap** is provided as an *implementation parameter* for the type **priority_queue**. Inclusion of the implementation parameter as well as the replacement of the generic pointer type **GenPtr** by the actual parameter types **int** and **node** is done by the interface in the file <LEDA/_prio.h>. See [MN95] for details.

```
⟨main.c 30⟩ ≡
  ⟨main includes 31⟩
  ⟨function dijkstra 29⟩
  int main(void)
  {
    GRAPH⟨int, int⟩ G;
    int n = read_int("#␣nodes␣=␣");
    int m = read_int("#␣edges␣=␣");
    random_graph(G, n, m);
    edge_array⟨int⟩ cost(G);
    node_array⟨int⟩ dist(G);
    node_array⟨edge⟩ pred(G);
    int M = read_int("max␣edge␣cost␣=␣");
    node s = G.choose_node();
    edge e;
    forall_edges (e, G) G[e] = cost[e] = rand_int(0, M);
    p_queue⟨int, node⟩ *PQ[2];
    PQ[0] = new p_queue⟨int, node⟩;
    PQ[1] = new _p_queue⟨int, node, r_heap⟩ (M);
    float T = used_time();
    float t_f = 0.0, t_r = 0.0;
    dijkstra(G, s, cost, dist, pred, *(PQ[0]));
    t_f = used_time(T);
    dijkstra(G, s, cost, dist, pred, *(PQ[1]));
    t_r = used_time(T);
    cout ≪ string("f_heap:␣%6.2f␣sec,␣r_heap:␣%6.2f␣sec\n", t_f, t_r);
  }
```

31. We need to include the following header files.

```
⟨main includes 31⟩ ≡
#include "r_heap.h"
#include <LEDA/random.h>
#include <LEDA/p_queue.h>
#include <LEDA/_p_queue.h>
#include <LEDA/graph.h>
#include <LEDA/graph_alg.h>
#include <fstream.h>
#include <string.h>
#include <math.h>
```

This code is used in websection 30.

32. Experiments.

We ran the experiments on a SUN workstation SPARC-10. The code was compiled with the GNU C++ compiler g++ and linked with the LEDA libraries lG (graphs) and lL (basis) and the math library lm. See the LEDA manual [Näh93] for more details on the use of the different LEDA libraries. The command line syntax to obtain the benchmark program is therefore

```
g++ -O -o r_heap r_heap.c main.c -lG -lL -lm
```

The test results are shown in tables. Additionally, they are visualized using GNUPLOT. Our inputs are generated using a LEDA facility that produces random graphs. An arbitrary node of the test graph is chosen as the start node s. Note that Dijkstra's algorithm as described above only visits the nodes reachable from s. In order to avoid biased results, we add a minimal number of edges to an input graph such that every node is reachable from s.

In our experiments, we varied the maximal edge costs C as well as the number of nodes and edges. We mainly examined sparse graphs since we were interested in problem instances where the theoretical time bounds for the implementations—$O(m + n \log n)$ vs. $O(m + n \log C)$—are actually different.

We ran two kinds of experiments. In the first kind, we measured the running time against varying C for three problems where n and m are fixed. The test cases are (1) $n = 1000, m = 10000$, (2) $n = 1000, m = 100000$ and (3) $n = 10000, m = 100000$.

In the second kind of tests, we measured the running time against varying n for four problems where C is fixed and $m = n\,f(n)$ for a given function $f(n)$. More specifically, for C we tested one small value, 10, and one large value, 10^6. For the number of edges m, we also chose one small value, $3n$, and one large value, $n \log n$. Note that we consider $m = n \log n$ as large although there might be $\Omega(n^2)$ edges. The reason is that for $m \gg n \log n$, choosing among the different heap implementations will no longer have a significant influence on the running time.

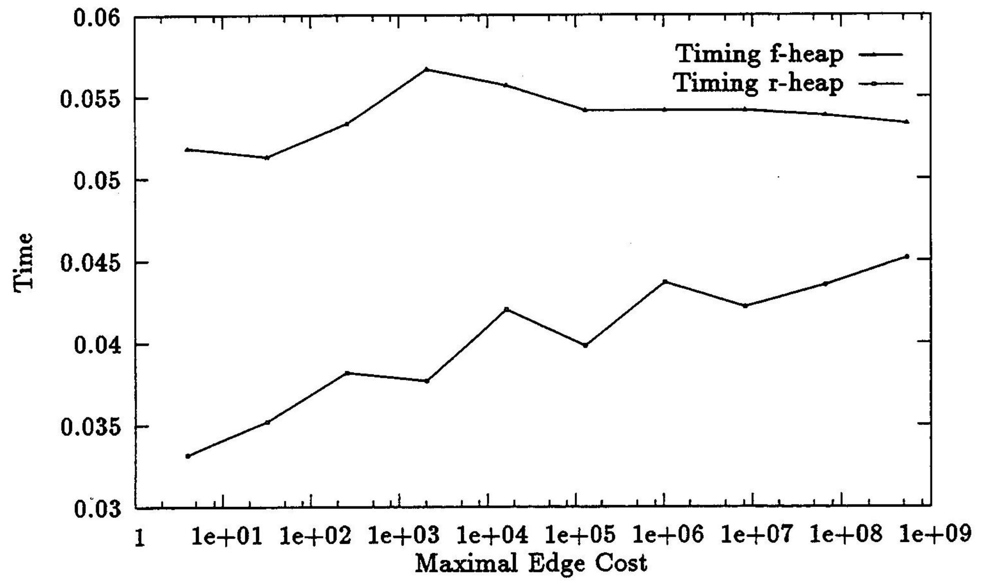
For each experiment, there is a table showing the running times t_f and t_r for Dijkstra's algorithm using **f_heap** and **r_heap**, respectively. These values are also shown in the GNUPLOT graph below the table. Additionally, the table lists the absolute and relative advantage of **r_heap** over **f_heap**, given by $t_f - t_r$ and $\frac{t_f - t_r}{t_r} \cdot 100$, respectively. All times are given in seconds.

33. Time vs Maximal Edge Cost C for $n = 1000$ and $m = 10000$.

The following test suite was run on a random graph with 1000 nodes and 10000 edges.

max. edge cost	time f_heap	time r_heap	diff.	%-Adv. of r_heap
$4 = 2^2$	0.051	0.032	0.019	56.280
$32 = 2^5$	0.050	0.035	0.016	45.972
$256 = 2^8$	0.052	0.037	0.014	39.737
2^{11}	0.056	0.037	0.019	50.442
2^{14}	0.055	0.041	0.013	32.539
2^{17}	0.054	0.039	0.013	35.983
2^{20}	0.054	0.044	0.011	24.045
2^{23}	0.054	0.041	0.012	28.458
2^{26}	0.054	0.043	0.009	23.756
2^{29}	0.052	0.045	0.008	18.079

Average Advantage of radix over fibonacci heaps in percent: 35.5296%

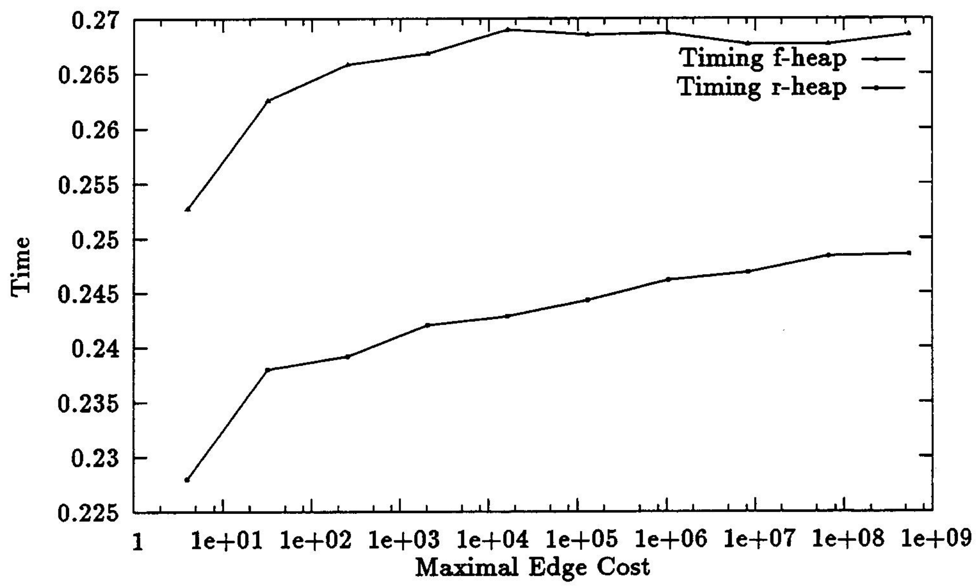


34. Time vs Maximal Edge Cost C for $n = 1000$ and $m = 100000$.

The following test suite was run on a random graph with 1000 nodes and 100000 edges.

max. edge cost	time f_heap	time r_heap	diff.	%-Adv. of r_heap
$4 = 2^2$	0.252	0.227	0.025	10.819
$32 = 2^5$	0.261	0.238	0.024	10.293
$256 = 2^8$	0.266	0.238	0.027	11.149
2^{11}	0.266	0.242	0.025	10.262
2^{14}	0.269	0.243	0.025	10.776
2^{17}	0.269	0.244	0.024	9.890
2^{20}	0.269	0.246	0.022	9.140
2^{23}	0.268	0.247	0.020	8.439
2^{26}	0.268	0.248	0.019	7.784
2^{29}	0.268	0.249	0.019	8.048

Average Advantage of radix over fibonacci heaps in percent: 9.66055%

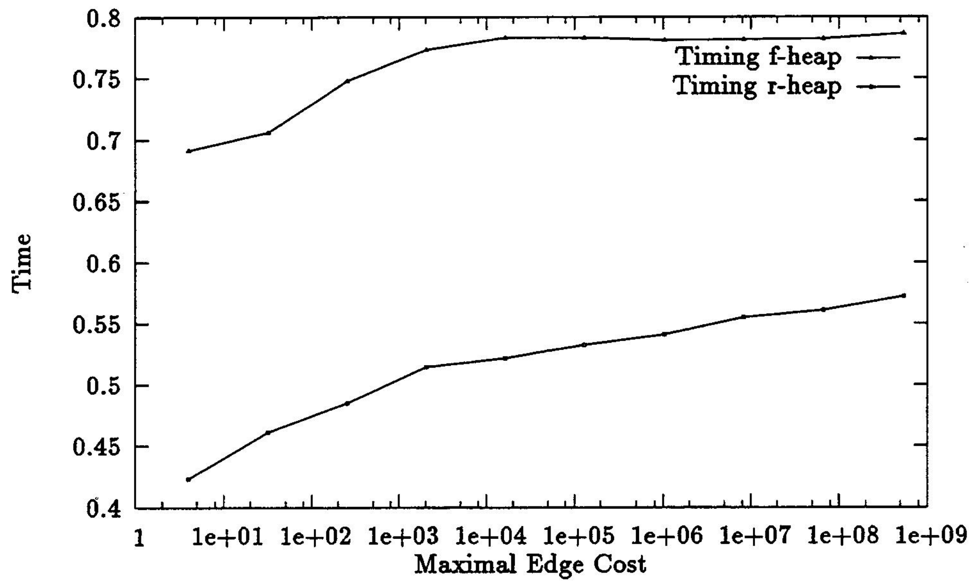


35. Time vs Maximal Edge Cost C for $n = 10000$ and $m = 100000$.

The following test suite was run on a random graph with 10000 nodes and 100000 edges.

max. edge cost	time f_heap	time r_heap	diff.	%-Adv. of r_heap
$4 = 2^2$	0.691	0.423	0.268	63.361
$32 = 2^2$	0.705	0.461	0.245	53.125
$256 = 2^2$	0.748	0.485	0.263	54.120
2^{11}	0.773	0.514	0.259	50.258
2^{14}	0.782	0.522	0.261	50.111
2^{17}	0.782	0.532	0.250	46.890
2^{20}	0.781	0.542	0.239	44.290
2^{23}	0.781	0.555	0.225	40.768
2^{26}	0.782	0.560	0.221	39.478
2^{29}	0.786	0.572	0.214	37.442

Average Advantage of radix over fibonacci heaps in percent: 47.9847%

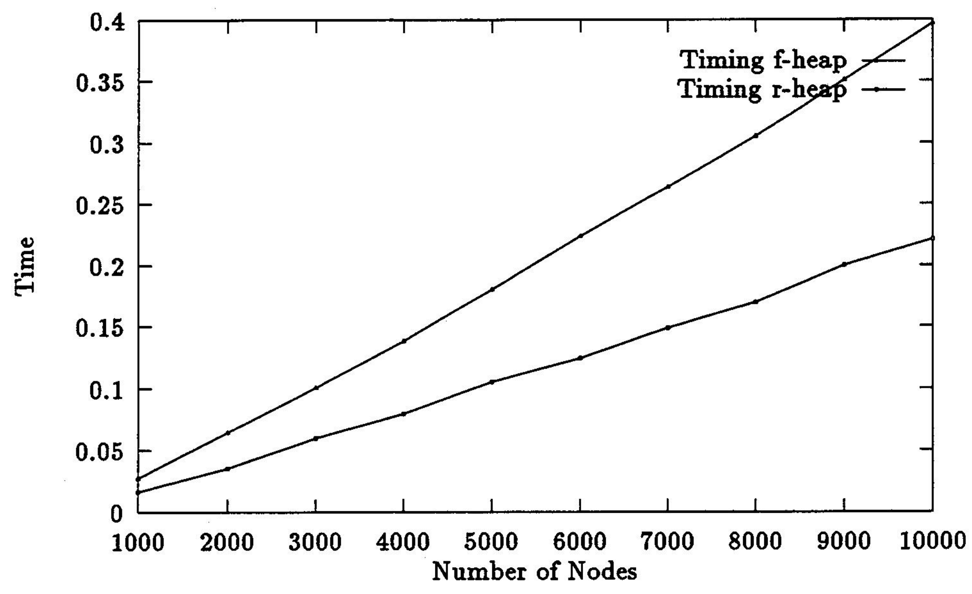


36. Time vs number of nodes n for $C = 10$ and $m = 3n$.

The following table shows test runs with maximal edge costs of 10. The test suite consists of random graphs with n nodes and $3n$ edges.

number of nodes	time f_heap	time r_heap	diff.	%-Adv. of r_heap
1000	0.027	0.017	0.011	64.706
2000	0.064	0.035	0.028	79.535
3000	0.100	0.059	0.041	67.688
4000	0.137	0.079	0.059	73.793
5000	0.180	0.104	0.075	71.472
6000	0.224	0.125	0.098	79.440
7000	0.263	0.149	0.114	77.466
8000	0.305	0.170	0.136	79.783
9000	0.351	0.200	0.150	75.478
10000	0.397	0.221	0.175	79.158

Average advantage of radix over fibonacci heaps in percent: 74.8525%

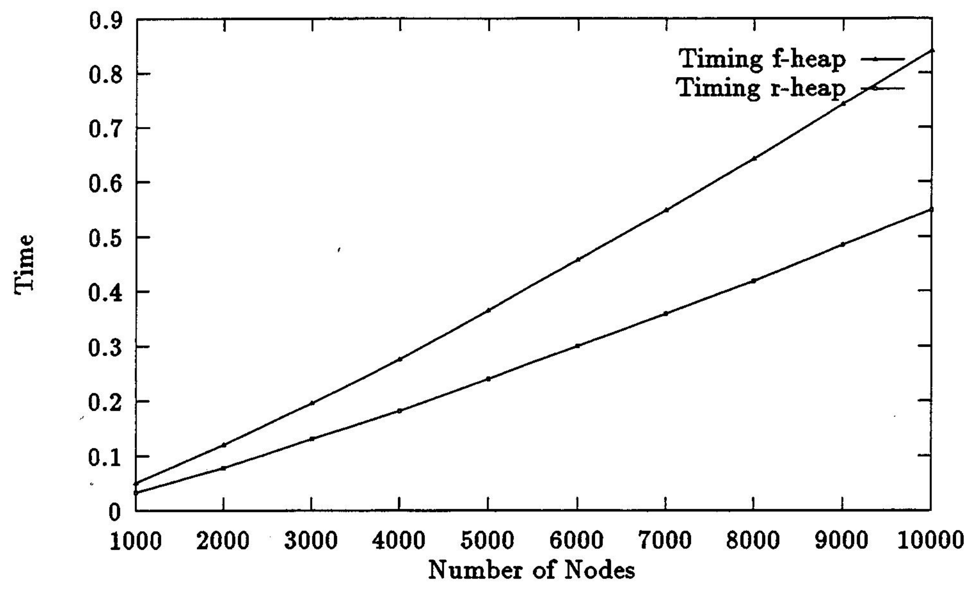


37. Time vs number of nodes n for $C = 10$ and $m = n \log n$.

The following table shows test runs with maximal edge costs of 10. The test suite consists of random graphs with n nodes and $n \log n$ edges.

number of nodes	time f_heap	time r_heap	diff.	%-Adv. of r_heap
1000	0.050	0.032	0.017	54.546
2000	0.119	0.078	0.041	54.272
3000	0.195	0.130	0.065	50.000
4000	0.275	0.180	0.095	52.719
5000	0.365	0.239	0.125	52.256
6000	0.456	0.300	0.157	52.110
7000	0.546	0.358	0.187	52.460
8000	0.639	0.418	0.222	53.068
9000	0.742	0.484	0.257	53.219
10000	0.839	0.549	0.291	52.958

Average advantage of radix over fibonacci heaps in percent: 52.761%

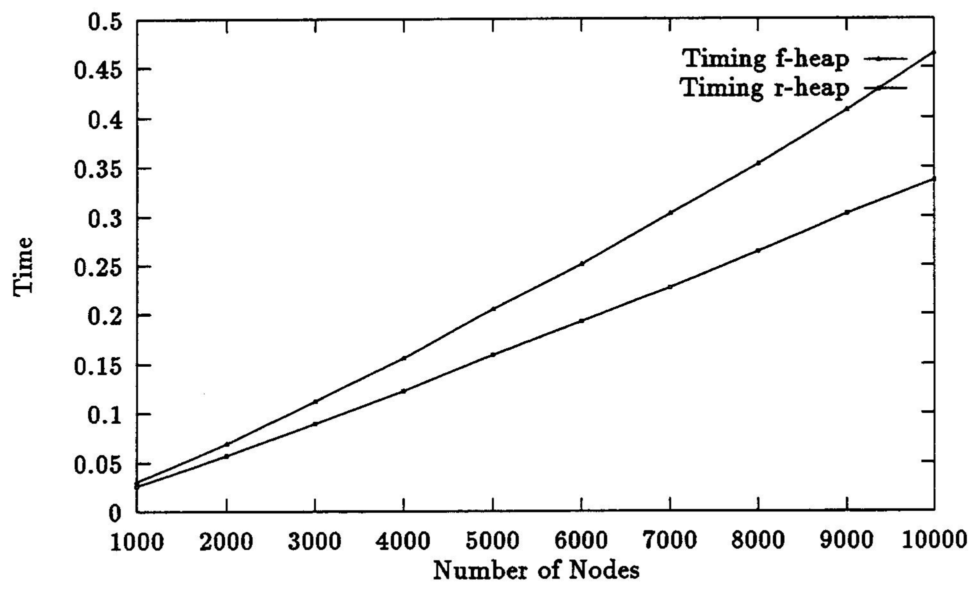


38. Time vs number of nodes n for $C = 10^6$ and $m = 3n$.

The following table shows test runs with maximal edge costs of 1000000. The test suite consists of random graphs with n nodes and $3n$ edges.

number of nodes	time f_heap	time r_heap	diff.	%-Adv. of r_heap
1000	0.030	0.025	0.004	17.721
2000	0.069	0.056	0.012	20.118
3000	0.112	0.088	0.022	25.794
4000	0.156	0.122	0.034	27.694
5000	0.204	0.158	0.045	28.961
6000	0.250	0.193	0.057	30.104
7000	0.302	0.226	0.075	33.138
8000	0.351	0.263	0.088	33.776
9000	0.407	0.301	0.104	34.804
10000	0.465	0.335	0.128	38.157

Average advantage of radix over fibonacci heaps in percent: 29.0267%

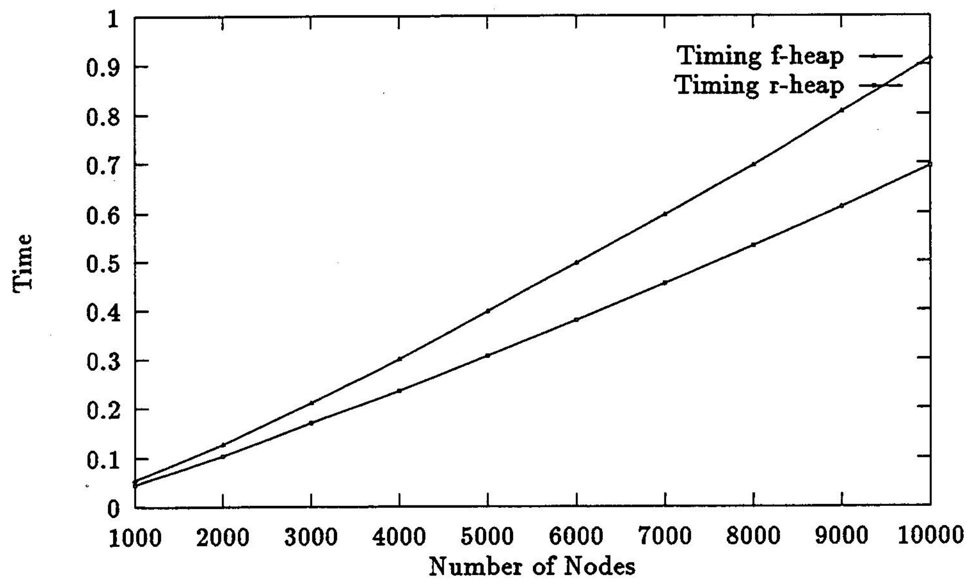


39. Time vs number of nodes n for $C = 10^6$ and $m = n \log n$.

The following table shows test runs with maximal edge costs of 1000000. The test suite consists of random graphs with n nodes and $n \log n$ edges.

number of nodes	time f_heap	time r_heap	diff.	%-Adv. of r_heap
1000	0.052	0.045	0.008	18.586
2000	0.126	0.102	0.024	23.625
3000	0.210	0.170	0.041	24.558
4000	0.300	0.236	0.064	27.368
5000	0.398	0.307	0.090	29.673
6000	0.496	0.379	0.114	30.337
7000	0.595	0.455	0.141	30.902
8000	0.694	0.531	0.164	30.865
9000	0.805	0.611	0.193	31.624
10000	0.913	0.694	0.219	31.635

Average advantage of radix over fibonacci heaps in percent: 27.9178%



40. Implementation notes.

This paper emerged from a semester project associated with a course on data structures and algorithms taught by Kurt Mehlhorn and Christian Schwarz at the Univerisität des Saarlandes in the winter semester of 1994/95, with teaching assistants Christoph Schmitz and Frank Schulz.

Given the specification r_heap.h, and the application framework consisting of the code for Dijkstra's algorithm, the test program and some test graphs, the students were asked to implement **r_heap** and to conduct a performance comparison against **f_heap** using the test program. The students were supposed to form small groups for this task.

Combining our own ideas with those provided by the solutions of the student teams and those resulting from discussions on the project, we came up with the final solution that is presented in this paper. Our solution is mainly based on work of the group formed by Jochen Könemann, Arnd Christian König and Mirek Riedewald.

Additionally, the contributions of all other students who worked on the project helped to shape this final version. These students are Klaus Brümmer, Claas Buchterkirche, Silvio Engel, Stefan Funke, Michael Gillmann, Ralf Glutting, Wolfgang Groß, Jan Holger Schmidt, Stephan Jost, Björn Kettner, Gunnar Klar, Gints Klavins, Boris Koldehofe, Klaus Kursawe, Karsten Kwappik, Markus Lentes, Peter Leven, Marc Meidlinger, Martin Reinstädtler, Andreas Schäfer, Ulrich Schmitt, Jan-Georg Smaus, Dirk Wagner, René Weiskircher and Frank Wittig. We thank them all.

41. References.

[MN95] K. Mehlhorn, S. Näher. *The LEDA platform for combinatorial and geometric computing.* In preparation.

[MN89] K. Mehlhorn, S. Näher. *LEDA: A library of efficient data types and algorithms.* LNCS, vol. 379, pp. 88-106, 1989, full version to appear in CACM.

[Näh93] S. Näher. *LEDA manual.* Technical report MPI-I-93-109, Max-Planck-Institut für Informatik, Saarbrücken, 1993.

[AMOT90] R. Ahuja, K. Mehlhorn, J. Orlin and R. Tarjan. *Faster Algorithms for the shortest path problem.* Journal of the ACM, Vol.37, 1990, pp.213-223.

[FT87] M. Fredman and R.E. Tarjan. *Fibonacci heaps and their uses in improved network optimization algorithms.* Journal of the ACM, Vol.34, 1987, pp. 596-615.

[D59] E. Dijkstra. *A note on two problems in connexion with graphs.* Numer. Math. 1, 1959, pp. 269-271.

Index

List of Refinements

⟨ class r_heap 4 ⟩ Used in websection 6.
⟨ constructors 8, 11 ⟩ Used in websection 28.
⟨ find new minimum 22 ⟩ Used in websection 21.
⟨ function dijkstra 29 ⟩ Used in websection 30.
⟨ includes 5 ⟩ Used in websection 6.
⟨ main includes 31 ⟩ Used in websection 30.
⟨ `main.c` 30 ⟩
⟨ private functions 9, 10, 12, 16 ⟩ Used in websection 28.
⟨ public functions 14, 15, 17, 18, 19, 20, 21, 24, 25, 26 ⟩ Used in websection 28.
⟨ r_heap includes 27 ⟩ Used in websection 28.
⟨ r_heap_node 3 ⟩ Used in websection 6.
⟨ `r_heap.c` 28 ⟩
⟨ `r_heap.h` 6 ⟩
⟨ reorganize r_heap 23 ⟩ Used in websection 21.