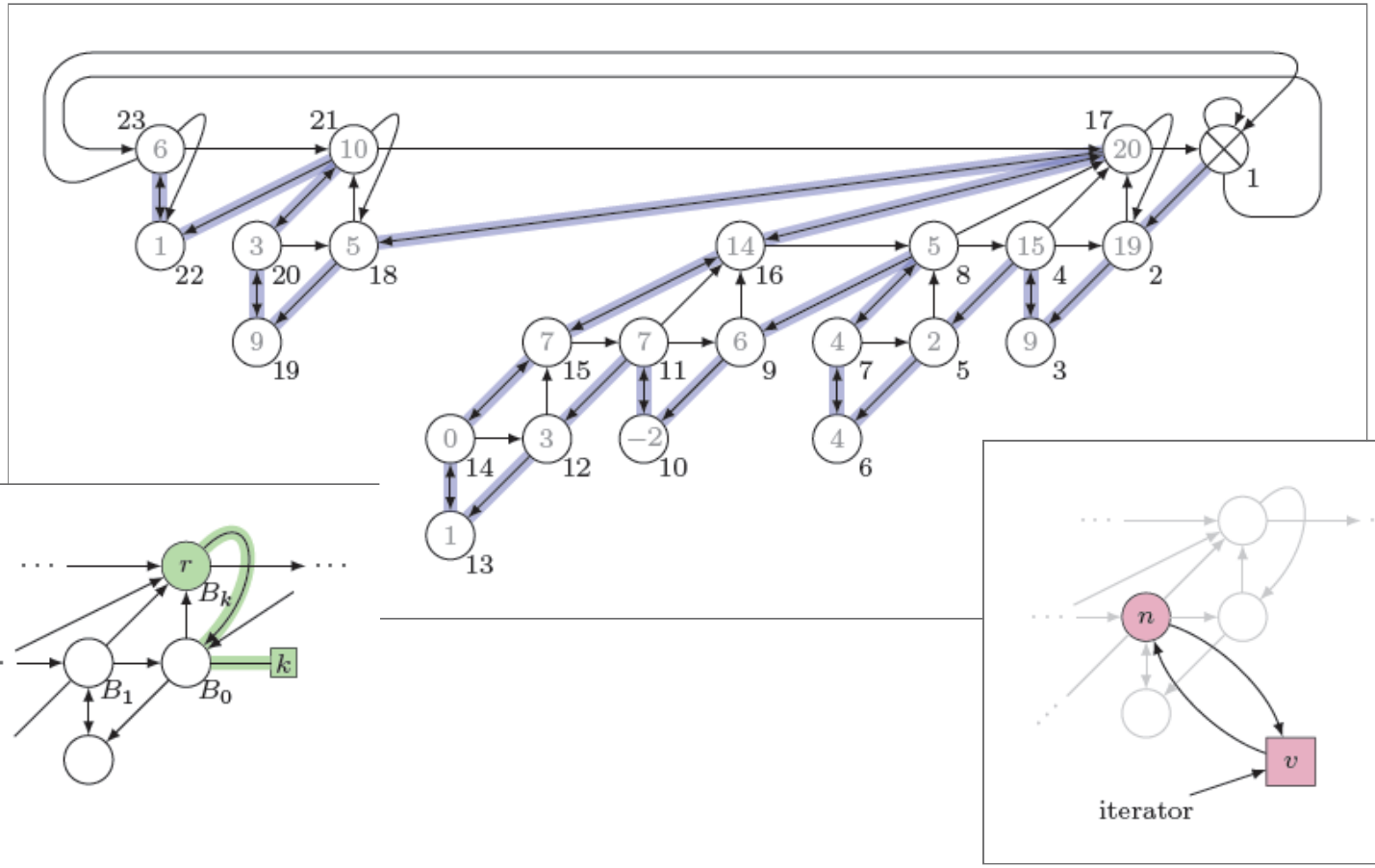
DATALOGISK INSTITUT
KØBENHAVNS UNIVERSITET

Miniprojekt:
"A meldable, iterator-valid priority queue"

En forbedret prioritetskø baseret på en binomial hob

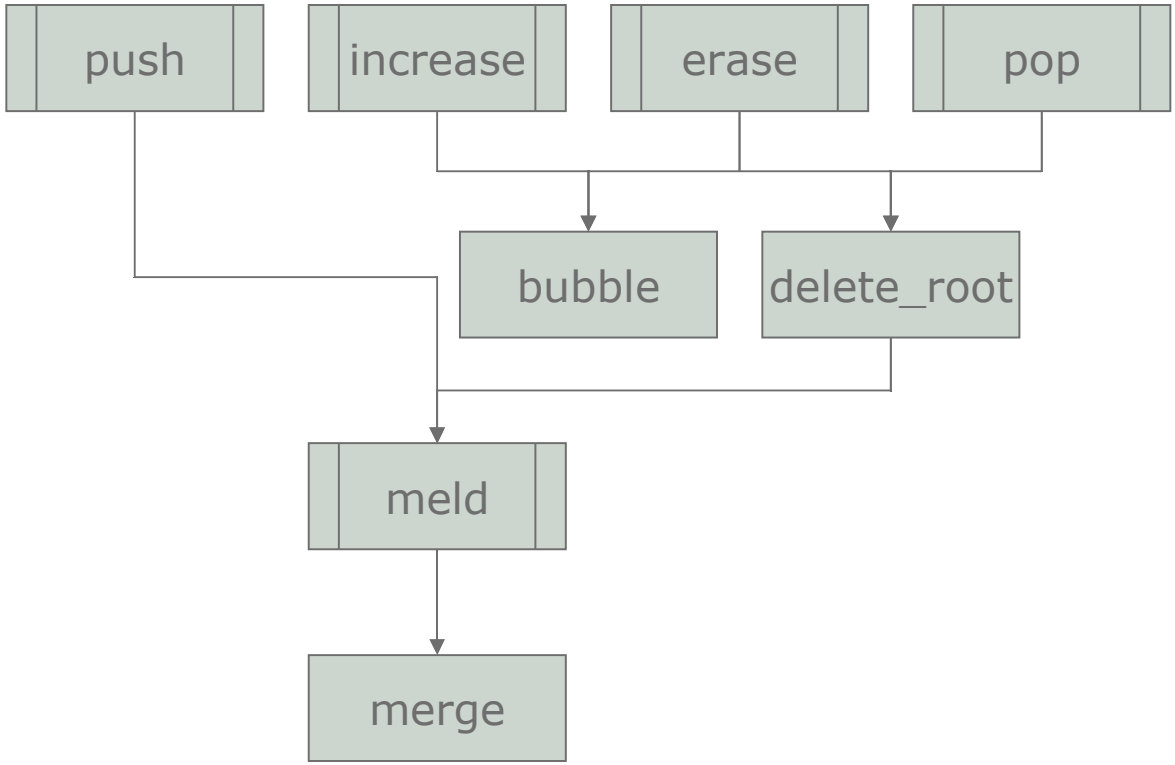
Resumé ...


23
6
21
10
17
20
1
1
22
3
20
5
18
14
16
5
8
15
4
19
2
9
19
7
15
7
11
6
9
4
7
2
5
9
3
0
14
3
12
−2
10
4
6
1
13
r
B_k
k
B_1
B_0
n
v
iterator

Fokus: Implementation

- Funktioner
  - Overblik
  - bubble og funktioner som parametre
  - Boost's function
- Property Maps
  - Generitet
  - Read/write
  - Boost's property maps
- Iteratorer
  - Bidirectional iterators
  - Const iterators
  - Boost's iterators
- Exceptions
  - Exception propagation og deallokering

Funktioner
Overblik – subfunktioner



Funktioner
bubble – brug af boost::function

```
template <typename V, typename C, typename A, typename N>
void binomial_heap<V,C,A,N>::bubble(
    boost::function<bool (container_type*,node_type*)>& predicate,node_type* x) {
USE_HEAP_PROPERTY_MAPS(N)
node_type* p = parent_r[x];
  while( predicate(this,x) )  {
    std::swap(node_ptr_w[value_node_ptr_r[x]],node_ptr_w[value_node_ptr_r[p]]);
    std::swap(value_node_ptr_w[x], value_node_ptr_w[p]);
    x = p;
    p = parent_r[x];
  }
}
```

Funktioner
bubble – brug af boost::function

```
template <typename V, typename C, typename A, typename N>
bool binomial_heap<V,C,A,N>::increase_predicate(node_type* x) {
  USE_HEAP_PROPERTY_MAPS(N)
  return  !is_root[x] && this->comparator(value_r[parent_r[x]], value_r[x]);
}
```

Bruges i **increase** på følgende måde:

```
// ...
boost::function<bool (container_type*,node_type*)> predicate;
predicate = &container_type::increase_predicate;
bubble(predicate,x);
// ...
```

Property Maps Generitet

Let udskiftning af implementation samtidigt med at syntaksen holdes simpel – algoritmerne i public methods skal ikke ændres.

... men property maps skal stadig have samme navne, og de ligger meget tæt op ad den valgte datastruktur.

Property Maps
boost::property_maps

```
template <typename Node>
struct degree_map :
  public boost::put_get_helper<int, degree_map<Node> > {  // CRTP
  typedef Node* key_type;
  typedef int value_type;
  typedef value_type& reference;
  typedef boost::lvalue_property_map_tag category;
  reference operator[](key_type const n) const {
    parent_map<Node> parent;
    sibling_map<Node> sibling;
    value_map<Node> value;
    child_map<Node> child;
    return (reference)sibling[parent[n]];
  }
};
```

Property Maps
lvalue / rvalue

Vi har et problem med **degree**, som har forskellig betydning afhængigt af om den er lvalue eller rvalue.

"degree[x] = y;" vs "y = degree[x];"

Compileren kan *ikke* selv vælge mellem

operator[](...) og operator[](...) const

ud fra lvalue/rvalue placering – det vælges ud fra om objektet, som operator[] er en method af, er const eller ej.

Vi kan altså **ikke** overloade operator[] afhængigt af om den er lvalue eller rvalue.

Property Maps
lvalue / rvalue

- Løsning: Vi laver et read og et write property map:
  degree[...] → **degree_w**[...] og **degree_r**[...]

- For at foregribe en anden implementation, hvor det ikke kun er degree, som kan have forskellig semantik afhængigt af lvalue/rvalue, gøres dette for alle property maps hvor vi både skal kunne læse fra og skrive til.

Property Maps
lvalue / rvalue

```
template <typename Node>
struct degree_map :
  public boost::put_get_helper<int, degree_map<Node> > {
  typedef Node* key_type;
  typedef int value_type;
  typedef value_type& reference;
  typedef boost::lvalue_property_map_tag category;
  reference operator[](key_type const n) const {
    parent_map<Node> parent;
    sibling_map<Node> sibling;
    value_map<Node> value;
    child_map<Node> child;
    return (reference)sibling[parent[n]];
  }
};
```

Property Maps
lvalue / rvalue

```
template <typename Node>
struct degree_map_readonly :
  public boost::put_get_helper<int, degree_map<Node> > {
  typedef Node* key_type;
  typedef int value_type;
  typedef value_type& reference;
  typedef boost::readable_property_map_tag category;
  value_type operator[](key_type n) const {
    parent_map<Node> parent;
    sibling_map<Node> sibling;
    child_map<Node> child;
    value_map<Node> value;
    if(child[n]) return (value_type)sibling[parent[n]];
    else return 0;
  }
};
```

Iteratorer
boost::iterator

Pga. vores datastrukturs specifikke design må vi implementere vores egen iterator.

- Vi vil gerne sikre, at vores iteratorer overholder STLs grænseflade.
- Vi vil gerne sikre, at brugerne af vores priority queue *ikke* kan bruge iteratorerne til direkte at ændre i datastrukturen.
- Vi vil gerne have letlæselig kode.

Dette kan opnås vha. boost::iterator.

Iteratorer
CRTP og STL egenskaber

```
template </* pmaps */>
class bidirectional_iterator : public boost::iterator_facade<
bidirectional_iterator</* pmaps */>,
typename Value_map::value_type const, boost::bidirectional_traversal_tag>
{
    typedef typename Succ_map::key_type key_type;
    typedef typename key_type::node_type node_type;
    typedef bidirectional_iterator</* pmaps */> iterator_type;
    typedef typename Value_map::value_type value_type;
    // ...
private:
    friend class boost::iterator_core_access;
    template </* pmaps */>
    bool equal(bidirectional_iterator </* pmaps */> const& other) const;
    void increment();
    void decrement();
    value_type const& dereference() const;
}
```

Iteratorer
Non-mutable egenskab

```
template </* pmaps */>
class bidirectional_iterator : public boost::iterator_facade<
bidirectional_iterator</* pmaps */>,
typename Value_map::value_type const, boost::bidirectional_traversal_tag>
{
    typedef typename Succ_map::key_type key_type;
    typedef typename key_type::node_type node_type;
    typedef bidirectional_iterator</* pmaps */> iterator_type;
    typedef typename Value_map::value_type value_type;
    // ...
private:
    friend class boost::iterator_core_access;
    template </* pmaps */>
    bool equal(bidirectional_iterator </* pmaps */> const& other) const;
    void increment();
    void decrement();
    value_type const& dereference() const;
}
```

Undtagelser

Vores kode er generisk, og gør stor brug af templates. Det er derfor ikke realistisk at tage højde for **alle** de undtagelser som templateparametrene evt. måtte kaste.

De fleste fanges således ikke af os og propagerer videre.

Men vi sørger for at

- fange og rydde op efter std::bad_alloc
- kaste meningsfulde undtagelser i vores public methods.

Undtagelser
std::bad_alloc

```
template <typename V, typename C, typename A, typename N>
binomial_heap<V,C,A,N>::binomial_heap(value_type const& v) :
    value_management(C(), A()) {
  // ...
  past_the_end = this->node_allocator.allocate(1);
  try  {
    past_the_end_v_node = this->value_node_allocator.allocate(1);
  }  catch(std::bad_alloc e)  {
    this->node_allocator.deallocate(past_the_end,1);
    throw e;
  }
  // fortsættes ...
```

Undtagelser
std::bad_alloc (fortsat)

```
 try  {
   m_head = this->node_allocator.allocate(1);
 }  catch(std::bad_alloc e)  {
   this->node_allocator.deallocate(past_the_end,1);
   this->value_node_allocator.deallocate(past_the_end_v_node,1);
   throw e;
 }
 try  {
   v_node = this->value_node_allocator.allocate(1);
 }  catch(std::bad_alloc e)  {
   this->node_allocator.deallocate(past_the_end,1);
   this->value_node_allocator.deallocate(past_the_end_v_node,1);
   this->node_allocator.deallocate(m_head,1);
   throw e;
 }
 // ...
};
```

Undtagelser

```
void increase(const_iterator iter, const value_type& new_value) {
  // ...
  if( new_value < *iter )
    throw std::out_of_range(AT " The new value does not increase the current value");
  // ...
}

const_iterator erase(const_iterator iter) {
  // ...
  x = iterator_node_r[iter];
  if( is_past_the_end[x] )
    throw std::out_of_range(AT " Trying to delete past the end node");
  // ...
}

void pop(){
  if(m_max)
    // ...
  else
    throw std::domain_error(AT " The queue is empty");
}
```

domain_error

logic_error